

NAVAL POSTGRADUATE SCHOOL

Monterey, California



19960801 073 **THESIS**

**RESEARCH ON MOTION PLANNING
OF
AUTONOMOUS MOBILE ROBOT**

by
Athanasios Papadatos
March 1996

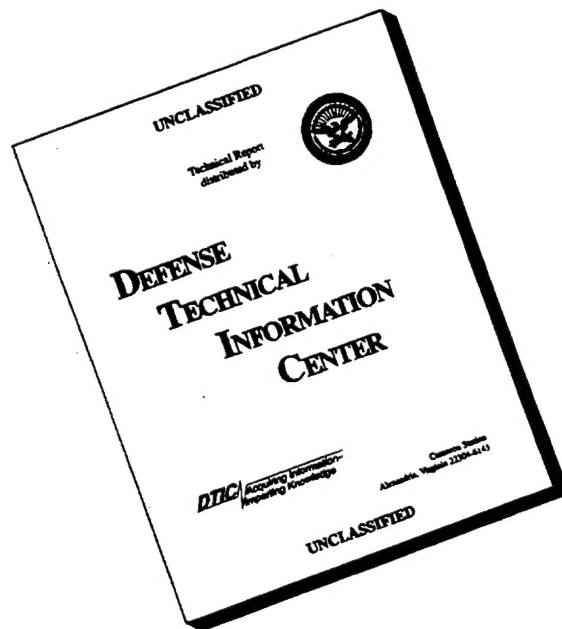
Thesis Advisor:
Thesis Co-Advisor:

Yutaka J. Kanayama
Xiaoping Yun

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
March 1996

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE

RESEARCH ON MOTION PLANNING OF AUTONOMOUS
MOBILE ROBOT(U)

5. FUNDING NUMBERS

6. AUTHOR(S)

Papadatos, Athanassios

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION
REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING
AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The path planning algorithm in Yamabico is based on a variation of Dijkstra's algorithm which has time complexity of $O(n^2)$. This algorithm works well in a dynamic environment, but a faster algorithm, called the All-Pairs Minimum Cost Paths algorithm, works even faster, $O(1)$, in the case of a static environment.

The computational complexity of the All-Pairs algorithm is $O(n^3)$, but if we know all pairs in advance, that is, the environment is static, we can preprocess them in advance, and use table lookup instead of Dijkstra's algorithm. Thus, we implemented a table lookup version for the static case, and kept Dijkstra's algorithm for the dynamic case. This results in both speed and flexibility.

This thesis also investigated the Linear Fitting Algorithm for Sonar testing. Range and angle data, from sonar, was fit to a straight line, giving resolution of 1 to 2.5 cm when the robot is within 100 to 150 cm of the line.

14. SUBJECT TERMS

Robotics, Sonar Testing, Global Motion Planning

15. NUMBER OF PAGES

116

16. PRICE CODE

17. SECURITY CLASSIFICATION

OF REPORT
Unclassified

18. SECURITY CLASSIFICATION

OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION

OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT

UL

Approved for public release; distribution is unlimited

**RESEARCH ON MOTION PLANNING
OF AUTONOMOUS MOBILE ROBOT**

Athanassios Papadatos
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of

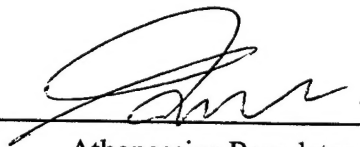
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

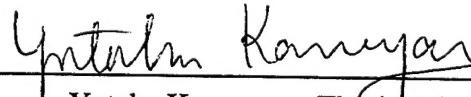
March 1996

Author:




Athanassios Papadatos

Approved by:



Yutaka Kanayama, Thesis Advisor



Xiaoping Yun, Thesis Co-Advisor



Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The path planning algorithm in Yamabico is based on a variation of Dijkstra's algorithm which has time complexity of $O(n^2)$. This algorithm works well in a dynamic environment, but a faster algorithm, called the All-Pairs Minimum Cost Paths algorithm, works even faster, $O(1)$, in the case of a static environment.

The computational complexity of the All-Pairs algorithm is $O(n^3)$, but if we know all pairs in advance, that is, the environment is static, we can preprocess them in advance, and use table lookup instead of Dijkstra's algorithm. Thus, we implemented a table lookup version for the static case, and kept Dijkstra's algorithm for the dynamic case. This results in both speed and flexibility.

This thesis also investigated the Linear Fitting Algorithm for Sonar testing. Range and angle data, from sonar, was fit to a straight line, giving resolution of 1 to 2.5 cm when the robot is within 100 to 150 cm of the line.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	OVERVIEW	1
1.	Sonar	1
2.	Global Motion Planning	2
II.	PROBLEM STATEMENTS	3
A.	SONAR CHARACTERISTICS MEASUREMENT	3
B.	GLOBAL MOTION PLANNING	4
III.	SONAR SYSTEM	7
A.	HARDWARE SYSTEM	7
1.	Sonar Grouping	8
2.	Range Finding	11
3.	Interrupt Control	11
4.	Data Transfer	12
B.	BASIC SONAR FUNCTIONS	12
1.	Distance	12
2.	Global Position Calculations	12
C.	LINEAR FEATURE EXTRACTION	14
1.	Least Squares Fitting	14
2.	Finding Endpoints	16
3.	Residual Testing	17
4.	Beginning Line Segments	17
5.	Ending Line Segments	18
IV.	MML USER INTERFACE	19
A.	GLOBAL CONFIGURATION CALCULATION	19
B.	SONAR FUNCTIONS	23
1.	Enable Sonar	23
2.	Disable Sonar	23
3.	Get Sonar Returns	23
4.	Get Global Sonar Returns	24
5.	Enable Linear Fitting	24
6.	Disable Linear Fitting	24
7.	Set Parameters In Linear Square Fitting	24
8.	Enable Data Logging	25
9.	Disable Data Logging	25
10.	Set Logging Interval	25
11.	Transfer Raw Data To Host	26
12.	Transfer Global Data To Host	26
13.	Transfer Segment Data To Host	26
C.	DATA LOGGING PROCEDURE	26
V.	SONAR CHARACTERISTICS EXPERIMENT RESULTS	29

A.	CASE 1	30
B.	CASE 2	31
C.	CASE 3	33
VI.	THEORY OF POLYGONS	37
A.	DISTANCE AND BISECTORS	37
B.	POLYGONAL WORLDS	38
1.	Polygons	38
2.	Subpolygons	41
3.	Distance from a Point to a Subpolygon	43
4.	Polygonal World	44
VII.	GLOBAL MOTION PLANNING / CONVEX DECOMPOSITION	47
A.	PATH CLASS	47
B.	DECOMPOSITION	48
C.	HOMOTOPIC DECOMPOSITION AND PATH REGION	51
D.	CONVEX DECOMPOSITION	54
VIII.	DIJKSTRA'S ALGORITHM	57
A.	CONNECTIVITY GRAPH OF THE WORLD	57
B.	ALGORITHM DESCRIPTION	60
C.	DATA STRUCTURE	61
D.	IMPLEMENTATION	62
IX.	ALL - PAIRS MINIMUM COST PATHS ALGORITHM	65
A.	CONNECTIVITY GRAPH OF THE WORLD	65
B.	ALGORITHM DESCRIPTION	69
C.	DATA STRUCTURE	71
D.	IMPLEMENTATION	71
X.	CONCLUSION	75
A.	RESULTS	75
1.	Sonar Characteristics Measurement	75
2.	Global Motion Planning	75
	APPENDIX A. USER PROGRAMMS	77
	APPENDIX B. PROGRAMMS' RESULTS	91
	LIST OF REFERENCES	103
	INITIAL DISTRIBUTION LIST	105

LIST OF FIGURES

Figure 1:	Robot's Motion in a Known World	3
Figure 2:	Motion Planning Problem	4
Figure 3:	Yamabico Sonars	7
Figure 4:	Sonar Hardware Architecture	8
Figure 5:	Composition	13
Figure 6:	Representation of a Line L Using r and a	15
Figure 7:	Example of First Compose	19
Figure 8:	Example of Compose with Sonar Return	21
Figure 9:	Result of Compose on Example	22
Figure 10:	Allowance in Linear Fitting Algorithm	29
Figure 11:	Case 1	30
Figure 12:	Case 1-Experiment Results	31
Figure 13:	Case 2	32
Figure 14:	Case 2-Experiment Results	33
Figure 15:	Case 3	34
Figure 16:	Case 3-Experiment Results	35
Figure 17:	Example World	38
Figure 18:	Normal Polygon	39
Figure 19:	Inverted Polygon	40
Figure 20:	Exterior Angle of a Convex Vertex	40
Figure 21:	Convex Polygons	41
Figure 22:	Concave Polygons	41
Figure 23:	A Rectilinear Polygonal World	42
Figure 24:	Subpolygon Decomposition of Concave Polygons	43
Figure 25:	Paths	48
Figure 26:	Decompositions	49
Figure 27:	Paths in a Decomposed World	50
Figure 28:	A Path which is not Regular	51
Figure 29:	Oriented Region	52
Figure 30:	Path Region	53
Figure 31:	Non-Orthogonal Border	55
Figure 32:	World Decomposition	57
Figure 33:	Basic Connectivity Graph	58
Figure 34:	Augmented Connectivity Graph	59
Figure 35:	Dijkstra's Algorithm	60
Figure 36:	Dijkstra's Data Structure	62
Figure 37:	Dijkstra's Example Execution	63
Figure 38:	A Decomposition and Its Basic Connectivity Graph	66
Figure 39:	Addition for Augmented Connectivity Graph	68
Figure 40:	All-Pairs Minimum Cost Paths algorithm	70
Figure 41:	Cost Array - Initial Matrix	72

Figure 42:	Previous Array - Initial Matrix	72
Figure 43:	Cost Array - Result Matrix	73
Figure 44:	Previous Array - Result Matrix.....	73

I. INTRODUCTION

A. BACKGROUND

One of the ultimate goals in robotics is to develop autonomous robots. Specifically, robots capable of successfully completing a task when provided instructions on what to do but not how to do it. Potential uses for sensor based robotics in structured environments abound. Increasingly capable autonomous mobile robot platforms are being developed to handle a myriad of hazardous duty assignments. While manufacturing tasks dominate the area of robotic applications, useful advances have been made in the areas of waste management, space exploration, undersea work, assistance for the disabled and medical surgery. Other examples include factory delivery systems, toxic waste disposal, pipeline inspections, etc. Several U.S. government-sponsored efforts are underway for building systems for military applications such as combating, handling ammunition, transporting material, underwater search and inspection operations, and other dangerous tasks currently performed by humans. Therefore, studying intelligent sensor-based systems is one of the major areas in the field of robotics today. In order to achieve a mission, sensors are used for data acquisition, a strategy based on the environment and state of the robot is chosen, and sensing is then integrated into the planning process. Hence, the global path planning system, local motion planning strategies, and sensing system are integrated into one coherent system.

B. OVERVIEW

1. Sonar

Sonars are used to determine location and recognize obstacles to be avoided. The sonar data are the inputs to several functions of a high level language called MML (model based mobile robot language), which is the driving force behind the robot *Yamabico*.

Although *Yamabico* may have precise knowledge of its location in a given environment, it is only capable of detecting the presence of unexpected obstacles in its path

using its 12 sonars. *Yamabico* can move at a speed up to 65 centimeters per second in a translational motion, forward or backwards.

In motion planning we attempt to apply the sonar system in parallel, so that combining the two techniques leads to a better “understanding” of the world and thus reduces the robot’s positional errors. Thus, sonar accuracy is very important as it affects the efficiency of motion planning and obstacle avoidance.

2. Global Motion Planning

Many of the robot’s tasks require motion. Deciding how to move from one location to another is known as the motion planning problem. In motion planning, not only is position important, but the orientation and curvature of the vehicle are important as it follows the path. These elements are represented in the *configuration* tuple (x, y, θ, κ) . Hence, the motion planning problem can be stated: How can a robot decide what motions to perform in order to move from one configuration to another?

Motion planning rather than *path planning* is used, because vehicles considered here are not points, but rigid bodies. In path planning, the result is a series of positions which must be followed by the vehicle. For an autonomous vehicle, planning motions which avoid known and unknown objects in its environment is the most fundamental functionality. For example, the mission of mine detection and clearance could not be successful unless the motion planning is solved. How should the data be structured to capture the topology of the operating environment, enable efficient processing of data, verify the robot location, and modify (if necessary) the global path plan? Consistency in both global and local plans can be achieved through constant collection and management of sensor data and using the data to guide the robot actions. All robot actions must be based on the status of the robot in conjunction with its environment.

II. PROBLEM STATEMENTS

A. SONAR CHARACTERISTICS MEASUREMENT

The Sonar Testing problem is to examine the precision of the sonar interpretation data, obtained by the current Linear Fitting Algorithm. Theoretical study, extensive simulations and testing are required to make this determination. The methodology of this thesis is to program the robot's motion in an known world, (Figure 1.), and then estimate how well it understands this world. The experiments concentrate primarily on the left and right sonars.

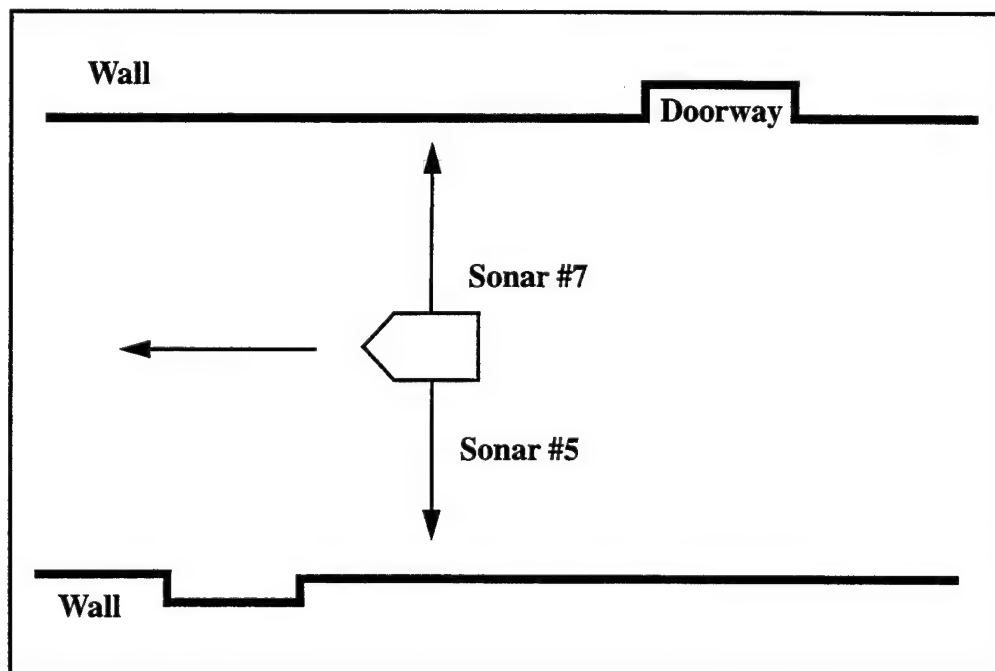


Figure 1. Robot's Motion in a Known World

Motion planning methods and algorithms need accurate data from the robot's sonar system. This enables verification that the robot is at the calculated position relative to the world's objects (polygons). If there is a difference between the actual and computed positions the software calculates the positional error.

The Linear Fitting Algorithm makes significant contributions toward solving the robot motion problem, (see Chapter V). Applying the theory in robotics, the sonar's returns are inputs to the algorithm, and the result is an interpretation of the sensed world compared to the known world. How well this combination of hardware and software resolves the disparity between the sensed world and known world is the criterion for the optimal sonar configuration.

B. GLOBAL MOTION PLANNING

The Global Motion Planning problem is, given the robot's operating environment (world), to finding the optimal path class.

The framework of the motion planning problem is set in a two dimensional, rectilinear world. The objective is to quickly and safely navigate an autonomous vehicle through free space from an initial configuration to a goal configuration using smooth motions (Figure 2.). A two dimensional, rectilinear world is used, since that type of world

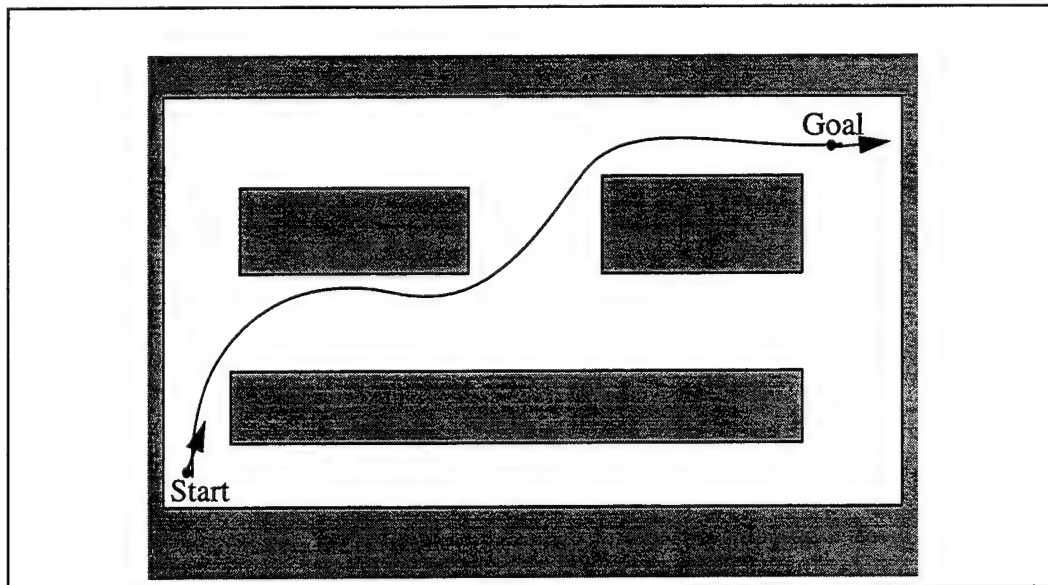


Figure 2. Motion Planning Problem

closely models the interior of most office buildings. A *rectilinear world* is one in which all edges are parallel to either the global coordinate frame x-axis or y-axis. Furthermore, all

obstacles are rectangles. A *configuration* (x, y, θ, κ) is a tuple which describes the two dimensional position, the orientation and the curvature of the vehicle. *Smooth motions* are those trajectories which possess continuous tangents and curvatures. *Free space* is that part of the world in which no obstacle is located.

The following assumptions are used throughout this thesis:

- The vehicle and all objects in the robot's environment are rigid bodies.
- The robot has complete knowledge of the environment in which it is operating. However, the use of external references to guide its motion other than the physical characteristics of the walls will not be used.
- The world is modeled as a planar rectilinear world with obstacles.
- The obstacles do not intersect or touch each other.
- Walls in the robot's environment are always rectilinear, as are found in most office buildings.
- All obstacles' faces are perpendicular to the plane in which the robot moves. This assumption is required to assure a good sensor return from all objects.
- Although the robot will be operating in a three dimensional environment, it is assumed that the model reflects the projection of the obstacles onto the plane of the floor on which the robot moves.
- All obstacles in the environment are immobile.
- Finally, it is assumed that the robot's free space is bounded by an inverted rectilinear polygon.

III. SONAR SYSTEM

A. HARDWARE SYSTEM

Yamabico's sonar hardware is extremely efficient because a dedicated sonar board with a microprocessor controls the sonar sensors [LOC94]. Yamabico's main central processing unit is interrupted only when data becomes available from the sonar array. The sonar system provides user interface functions that control Yamabico's array of sonar range finders. At any point within a user's program, any of the 12 sonars may be enabled or disabled. This allows the user to operate a given sonar only when necessary for a particular application. When needed, the sonar system returns the latest reading of a specified sonar out of the twelve. This system design is far better than the primitive one in which a user must wait 30 milliseconds after he/she issues a command. A user's program can also be forced to "busy wait" until some sonar-based condition is satisfied. This feature is particularly valuable for obstacle avoidance. For example, a user's program could be written to wait until the forward looking sonar's range is less than distance d , then stop. A block diagram of the sonar system is provided in Figure 3.

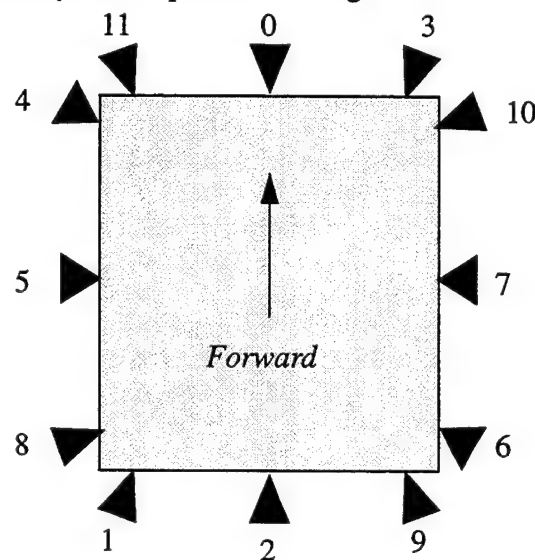


Figure 3. Yamabico Sonars

Figure 4. shows the current hardware configuration of *Yamabico*.

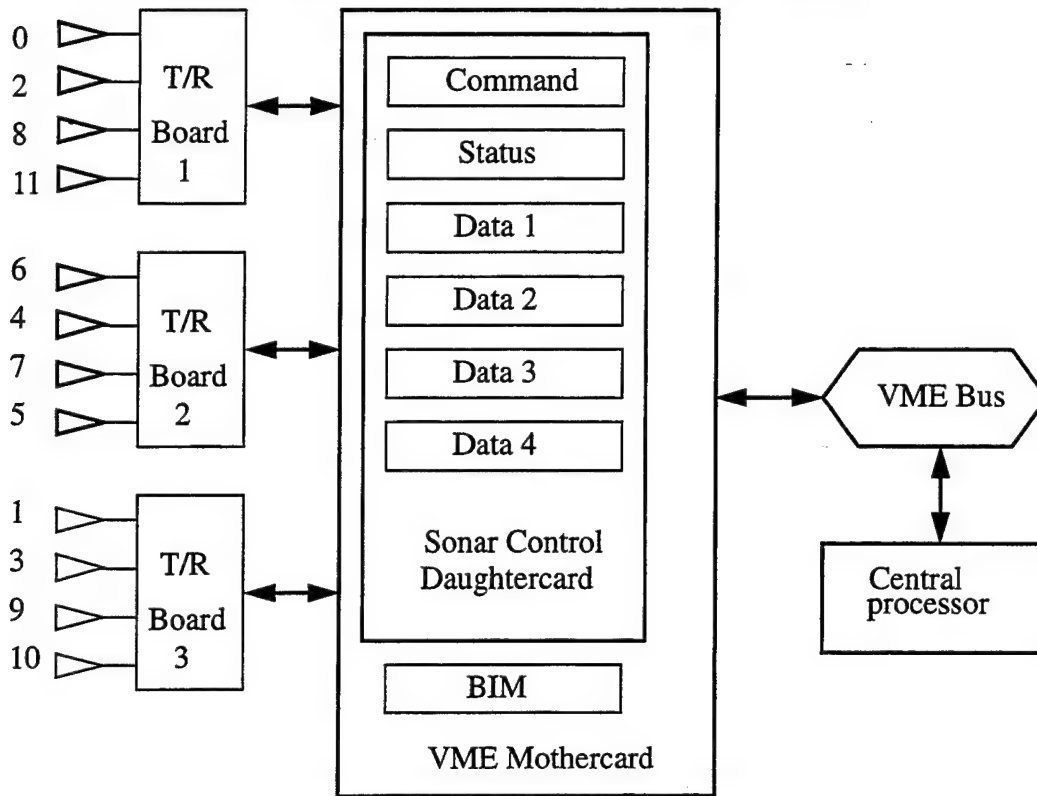


Figure 4. Sonar Hardware Architecture

1. Sonar Grouping

In order to reduce sampling time the sonars are operated in logical groups of four. The sonars of a logical group are all pulsed simultaneously and thus the sampling time is reduced by a factor of four as compared to individual firing of the sonars. The sonars of each logical group are oriented in such a way as to:

- prevent mutual interference
- provide a "look" in all four directions from each group
- present a similar aspect from each sonar during a rotational scan

Thus, logical group 0 consists of sonars 0, 2, 5 and 7 (see Figure 3.), group 1 consists of sonars 1, 3, 4 and 6; group 2 consists of sonars 8, 9, 10 and 11; (see page 9) and

group 3 is a “virtual” group which consists of four permanent test values. The sonars of a group are symmetric about the robot’s axis of rotation.

TABLE 1: Sonar Mnemonics

Mnenonic	Sonar	Group
S000	0	0
S030	3	1
S060	10	2
S090	7	0
S120	6	1
S150	9	2
S180	2	0
S210	1	1
S240	8	2
S270	8	2
S300	4	1
S330	11	2

In addition to being *logically* grouped, the sonars are also *physically* grouped (see Figure 4.). The physical grouping of the sonars is made to distribute the electrical load over the driver boards evenly and thus minimize any electrical transients associated with operation of the sonar. The physical grouping connects sonars 0, 2, 8 and 11 to driver/amplifier board 1; sonars 4, 5, 6 and 7 to board 2; and sonars 1, 3, 9 and 10 to board 3. The reader will note that pairs of sonars from logical groups are assigned to physical groups, for example, sonars 0 and 2 from logical group 0 are assigned to physical group (driver/amplifier board) 1.

Initial design of the control circuitry was based on two primary parameters [BYR94]: (1) a desired maximum range of 400 centimeter and (2) a pulse width of 1 millisecond. Assuming a speed of sound in air, at sea level, of 340 meters/second we may calculate a round-trip time:

$$\text{round trip time} = \frac{400 \text{ cm}}{34000 \text{ cm/sec}} \times 2 = 23.53 \text{ msec} \quad (\text{Eq. 1})$$

This round trip time is the period during which a valid echo may be received and is referred to as the *receive gate*. This interval is rounded up to 24 milliseconds and is derived by division of the sonar system's 2 MHz clock to ensure that the receiver is not falsely triggered by a direct path reception from it's adjacent transmitter, we opt to disable the receiver until the transmit pulse is complete. This will have the disadvantage of setting a minimum range equal to half the distance sound would travel in the time of a transmit pulse.

$$\text{minimum range} = 34000 \text{ cm./sec.} \times 1 \text{ msec.} \times 0.5 = 17 \text{ cm.} \quad (\text{Eq. 2})$$

This minimum range lies approximately 9 centimeters outside the periphery of the robot. In order to allow the measurement of objects up to the periphery of the robot, the pulse width was decreased to 0.5 milliseconds thus reducing the minimum range to 8.5 centimeters.

In actual practice, the minimum range is set by firmware to 9.6 centimeters, the additional distance being due to time allotted for switching and settling in the circuitry.

All sonars of a logical group are pulsed simultaneously. Which groups are fired is determined by the value of the corresponding bit in the *command register* of the sonar control board, which in turn is set by the user with an MML function (Figure 4.). Hence, if bit 2 is set to 1 then group 2 sonars will be pulsed. If more than one group is selected to be pulsed, the sonar control board will pulse the first group on the list, and when the data from that pulse has been read from the fourth data register the sonar control board will proceed to the next group and pulse it, and so on in round robin fashion. Groups with their control bit set to 0 will *not* be pulsed. The sampling rate can thus be as high as 41 Hz with only one group enabled (based on a 24 millisecond read gate as determined in Equation 3.2) and will be halved for each additional group enabled. At a nominal robot speed of 30 centimeters

per second. this sampling rate could provide an updated range within 0.75 centimeter of travel, exceeding our desired positional accuracy of 1 centimeter. Of course, real performance will be affected by any delay in reading the data registers due to other demands on the central processor (processing the sonar data, controlling motion, etc.).

2. Range Finding

There are four 16 bit data registers on the sonar control board, one for each of the four sonars in a logical group. When the transmit pulse is sent to the driver/amplifier boards a counter is started which increments each of the data registers every 6 microseconds. This time period is equivalent to a range of 1.02 millimeter:

$$\text{range} = 340000 \text{ mm/sec} \times 6 \text{ microsec} \times 0.5 = 1.02 \text{ mm} \quad (\text{Eq. 3})$$

The incrementing of a particular data register continues until an echo is received or the range gate times out. The first 12 bits of the data register are allotted for range accumulation, thus allowing for a maximum range of 4.177 meters ($4095 \times 1.02 \text{ mm}$). If the range gate should time out before an echo is received, the high bit of the over ranged sonar's data register is set to 1. This is the "over range" bit and is used to signal the ensuing software that no echo was received. Bits 12, 13 and 14 of the data registers are not used. When the ranging cycle is complete, the appropriate group number is written into bits 4 and 5 of the status register and the "ready" bit, bit 7 of the status register, is set to 1. The ready bit is used as a flag when operating in the polled mode; i.e. without interrupts.

3. Interrupt Control

The sonar control board is actually a daughtercard which rides on a VME bus mothercard. The mothercard carries address decoders, bus drivers and interrupt control circuitry in the Bus Interface Module (BIM).

When the sonar has completed a ranging cycle an interrupt request is provided to the BIM. The BIM's *control register* holds information which determines whether an interrupt is to be generated or not, and if so which interrupt level is to be generated. Presuming an interrupt is generated, when the correct acknowledgment returns on the address lines the BIM's *vector register* provides the vector table entry where the central processor may find the vector to the interrupt handler. The correct interrupt level, the

interrupt enable bit and interrupt vector are loaded to the BIM during software initialization.

4. Data Transfer

Each of the data registers is individually addressed on the VME bus by a VME short address, as is the status register. Transferral of the data is extremely straightforward. The interrupt handler simply reads the correct register, masks out the unwanted bits and writes the data to the stack. When the last data register is read, the sonar system resets the data registers and commences a ranging cycle on the next sonar group in it's round robin. The system will continue to operate autonomously until all the sonars are disabled.

B. BASIC SONAR FUNCTIONS

1. Distance

There are two functions available to return sonar values. One function, `sonar()` will return the range from the sonar to the object it is getting the return from. If there is no return, then a value of infinity is assigned, and for *Yamabico* this value is 999999. The infinity value is used for trouble shooting purposes, to detect whether or not there are instances of no return from objects at a distance of less than 4 meters. The second range function available is `global()`, and this will return the x,y coordinates of where the return was detected in the world that the vehicle is in. This is useful in the vehicle making a map of its world with obstacles in it.

2. Global Position Calculations

By utilizing the compose function seen in Figure 5., we can determine the actual point in a 2D coordinate system. Let the following equations represent q_1 and q_2 ,

$$q_1 = (x_1, y_1, \theta_1)^T \quad (\text{Eq. 4})$$

$$q_2 = (x_2, y_2, \theta_2)^T \quad (\text{Eq. 5})$$

The composition of these transformations is defined as

$$q_1 \circ q_2 \equiv \begin{pmatrix} x_1 + x_2 \cos \theta_1 - y_2 \sin \theta_1 \\ y_1 + x_2 \sin \theta_1 + y_2 \cos \theta_1 \\ \theta_1 + \theta_2 \end{pmatrix} \quad (\text{Eq. 6})$$

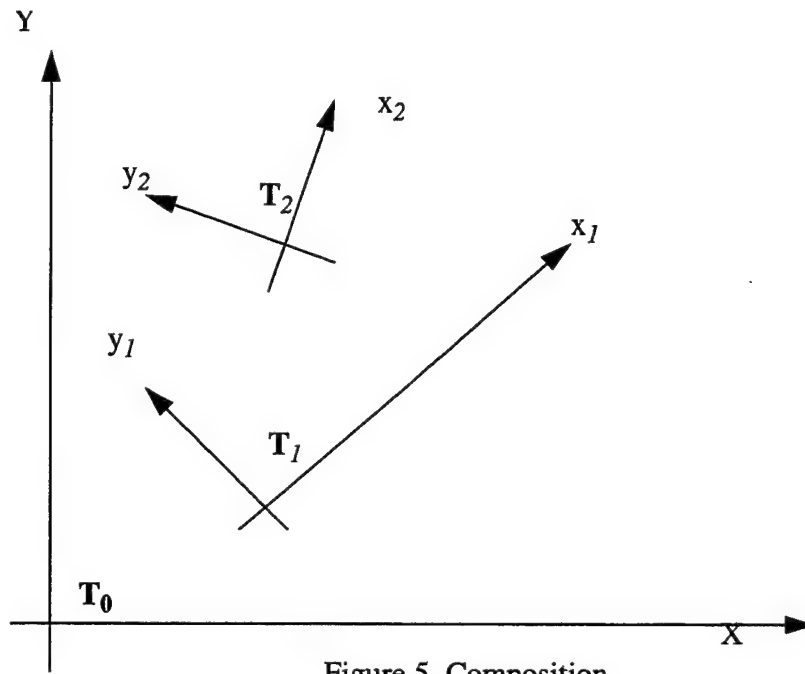


Figure 5. Composition

This functionality is extremely useful in dynamically configuring new paths from our original paths. We can dynamically define another path depending on your position and the direction of your vehicle. For the sonar functions, it allows much more modularity to the code. The code is reusable, since the only thing unique to *Yamabico* are the actual sonar positions on the robot.

C. LINEAR FEATURE EXTRACTION

In addition to simple *range* and *point position* data, the sonar system recognizes the *linear features* of an orthogonal world. To do so we must provide some method for recognizing sets of data points which form the linear feature and a method for finding and describing the line segment that best fits that set of data points. This is accomplished in reverse fashion, i.e. we presume the data we are receiving belongs to such a set and continuously modify a descriptive line segment to a best fit of the data using a least squares fitting algorithm. This line segment continues to grow until the incoming data or certain measures of the line segment indicate that the line segment should be ended and a new one started.

1. Least Squares Fitting

Suppose we have collected n consecutive valid data points in a local coordinate system, (p_1, \dots, p_n) , where $p_i = (x_i, y_i)$ for $i = 1, \dots, n$. We obtain the moments m_{jk} of the set of points

$$m_{jk} = \sum_{i=1}^n x_i^j y_i^k \quad (0 \leq j, k \leq 2, \text{ and } j+k \leq 2) \quad (\text{Eq. 7})$$

Notice that $m_{00} = n$. The *centroid* C is given by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \equiv (\mu_x, \mu_y) \quad (\text{Eq. 8})$$

The secondary moments around the centroid are given by

$$M_{20} \equiv \sum_{i=1}^n (x_i - \mu_x)^2 = m_{20} - \frac{(m_{10})^2}{m_{00}} \quad (\text{Eq. 9})$$

$$M_{11} \equiv \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \left(\frac{m_{10}m_{01}}{m_{00}} \right) \quad (\text{Eq. 10})$$

$$M_{02} \equiv \sum_{i=1}^n (y_i - \mu_y)^2 = m_{02} - \frac{(m_{01})^2}{m_{00}} \quad (\text{Eq. 11})$$

We adopt the parametric representation (r, α) of a line with constants r and α . If a point $p = (x, y)$ satisfies an equation

$$r = x \cos \alpha + y \sin \alpha \quad (-\pi/2 < \alpha \leq \pi/2) \quad (\text{Eq. 12})$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r (Figure 6.). This method has an advantage in expressing lines that are perpendicular to the X axis. The point-slope method, where $y = mx + b$, is incapable of representing such a case ($m = \infty$, b is undefined).

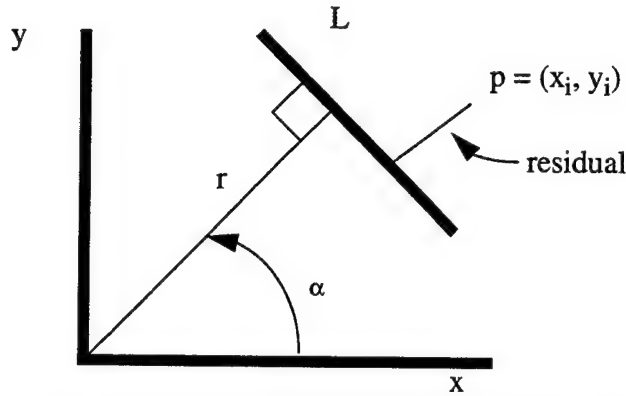


Figure 6. Representation of a Line L Using r and α

The residual of point $p_i = (x_i, y_i)$ and the line $L = (r, \alpha)$ is $x_i \cos \alpha + y_i \sin \alpha - r$. Therefore, the sum of the squares of all residuals is

$$S = \sum_{i=1}^n (r - x_i \cos \alpha - y_i \sin \alpha)^2 \quad (\text{Eq. 13})$$

The line which best fits the set of points is supposed to minimize S . Thus the optimum line (r, α) must satisfy

$$\frac{dS}{dr} = \frac{dS}{d\alpha} = 0 \quad (\text{Eq. 14})$$

Thus,

$$\frac{dS}{dr} = 2 \sum_{i=1}^n (r - x_i \cos \alpha - y_i \sin \alpha) \quad (\text{Eq. 15})$$

$$= 2 \left(r \sum_{i=1}^n 1 - \left(\sum_{i=1}^n x_i \right) \cos \alpha - \left(\sum_{i=1}^n y_i \right) \sin \alpha \right) \quad (\text{Eq. 16})$$

$$= 2 (rm_{00} - m_{10} \cos \alpha - m_{01} \sin \alpha) \quad (\text{Eq. 17})$$

$$= 0$$

and

$$r = \frac{m_{10}}{m_{00}} \cos \alpha + \frac{m_{01}}{m_{00}} \sin \alpha = \mu_x \cos \alpha + \mu_y \sin \alpha \quad (\text{Eq. 18})$$

where r may be negative. Substituting r in Equation 13 by Equation 18,

$$S = \sum_{i=1}^n ((x_i - \mu_x) \cos \alpha + (y_i - \mu_y) \sin \alpha)^2 \quad (\text{Eq. 19})$$

Finally,

$$\frac{dS}{d\alpha} = 2 \sum_{i=1}^n ((x_i - \mu_x) \cos \alpha + (y_i - \mu_y) \sin \alpha) (- (x_i - \mu_x) \sin \alpha + (y_i - \mu_y) \cos \alpha) \quad (\text{Eq. 20})$$

$$= 2 \sum_{i=1}^n \left((y_i - \mu_y)^2 - (x_i - \mu_x)^2 \right) \sin \alpha \cos \alpha + 2 \sum_{i=1}^n (x_i - \mu_x) (y_i - \mu_y) (\cos^2 \alpha - \sin^2 \alpha) \quad (\text{Eq. 21})$$

$$= (M_{02} - M_{20}) \sin 2\alpha + 2M_{11} \cos 2\alpha \quad (\text{Eq. 22})$$

$$= 0$$

Therefore

$$\alpha = \frac{\text{atan}(2M_{11} / (M_{02} - M_{20}))}{2} \quad (\text{Eq. 23})$$

Equation 18 and Equation 23 are the solutions for the line parameters generated by a least squares fit.

2. Finding Endpoints

The residual of a point $p_i = (x_i, y_i)$ is

$$\delta_i = (\mu_x - x_i) \cos \alpha + (\mu_y - y_i) \sin \alpha \quad (\text{Eq. 24})$$

Therefore, the projection, p'_i of the point p_i onto the major axis is

$$p'_i = (x_i + \delta_i \cos \alpha, y_i + \delta_i \sin \alpha) \quad (\text{Eq. 25})$$

We will use p'_1 and p'_n as estimates of the endpoints of the line segment L obtained from the set p of data points.

3. Residual Testing

We wish to do some pre-filtering of the data in order to remove points from the data stream which are clearly *not* colinear with the existing points of set p . In this way we can often detect the end of a line segment before having to perform the considerable computations necessary to include it in the line. If the point satisfies

$$\delta_{i+1} < \max(\sigma \times C1, C2) \quad (\text{Eq. 26})$$

where $C1$ and $C2$ are positive constants (typically, $C1 = 0.02$ and $C2 = 5.0$) then the point can be included in the current line segment. $C2$ at 5.0 allows for more residual at a distance greater than 250 centimeters, up to 8 centimeters at a distance of 4 meters.

4. Beginning Line Segments

First, the sonar returns must fall within their physical constraints. For Yamabico, acceptable return values fall between 9.3 centimeters and 409.0 centimeters. If a sonar return is not within this range, a segment will be generated if there have been at least 10 previous returns that met all requirements of the least square fitting to qualify as a segment.

Secondly, if it is the first return, you simply store it as the starting point and proceed with the next return.

With the line segment established, collection and testing of the additional data points can proceed. If the data point *passes* the residual testing, the moments and test values for the line are calculated including the new point. Should that test pass, the line segment parameters (endpoints, length, etc.) are updated and the system proceeds to gather a new data point.

5. Ending Line Segments

There are three ways in which a line segment is ended. It may be ended by the failure of data points to pass the residual testing, explicitly ended by the sonar being disabled, or by the sonar return being outside the acceptable range.

IV. MML USER INTERFACE

A. GLOBAL CONFIGURATION CALCULATION

The compose function is implemented in a sonar function called `calculate_global()`. It applies the compose function twice [BYR94]. The first time the compose function is used to determine the actual position of the sonar in the world being navigated by the vehicle, as seen in Figure 7. In this example Yamabico is at coordinates (80,40), in the “world coordinates”. The sonars position on the robot is (9.5, -19.75). By applying the compose

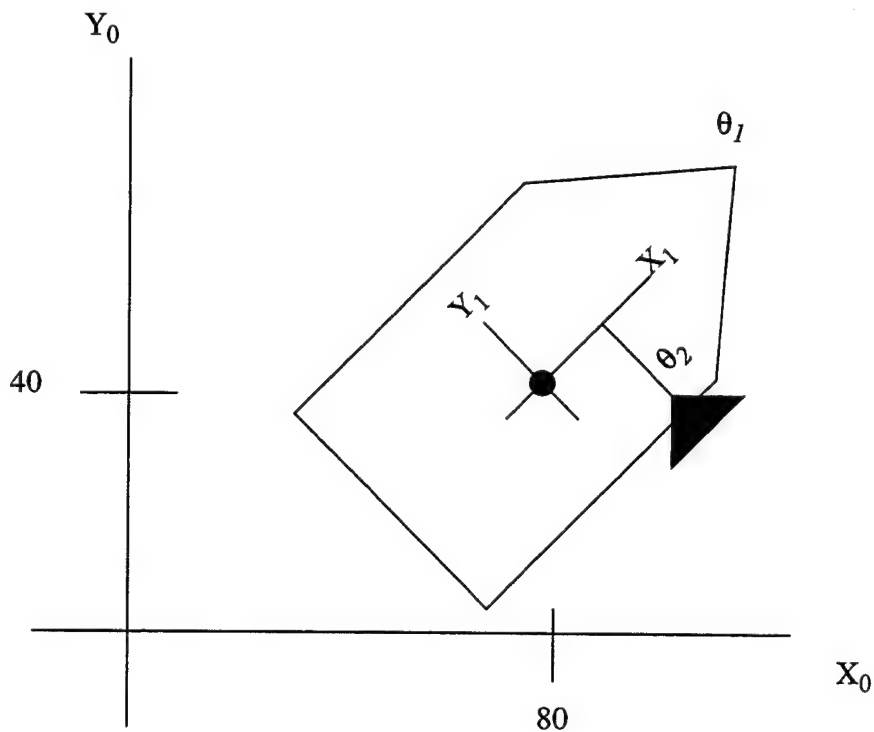


Figure 7. Example of First Compose

function,

$$q_1 \circ q_2 = \begin{pmatrix} x_1 + x_2 \cos \theta_1 - y_2 \sin \theta_1 \\ y_1 + x_2 \sin \theta_1 + y_2 \cos \theta_1 \\ \theta_1 + \theta_2 \end{pmatrix}$$

we determine the position of the sonar in “world coordinates”. In this case it would be:

$$\text{world sonar x coordinate} = 100.68 = 80 + 9.5 \cdot \cos(\pi/4) - (-19.75 \cdot \sin(\pi/4))$$

$$\text{world sonar y coordinate} = 32.74 = 40 + 9.5 \cdot \sin(\pi/4) + (-19.75 \cdot \cos(\pi/4))$$

$$\text{world sonar theta} = -\pi/4 = \pi/4 + -(\pi/2).$$

The second time compose is applied it determines where the sonar return is in the world being navigated by the robot, as in Figure 8.

In this case we apply the compose function and the results are:

$$\text{sonar x coordinate from robot} = 171.42 = 100.68 + 35 \cdot \cos(-\pi/4) - 0 \cdot \sin(-\pi/4)$$

$$\text{sonar y coordinate from robot} = -37.94 = 32.74 + 35 \cdot \sin(-\pi/4) + 0 \cdot \cos(-\pi/4)$$

which gives us the point in Figure 9.

By knowing where each sonar is on the vehicle and knowing where the vehicles position is, we can consistently determine where the object being detected is in relation to the world that Yamabico is in. This is needed so that a vehicle can dynamically map out the world.

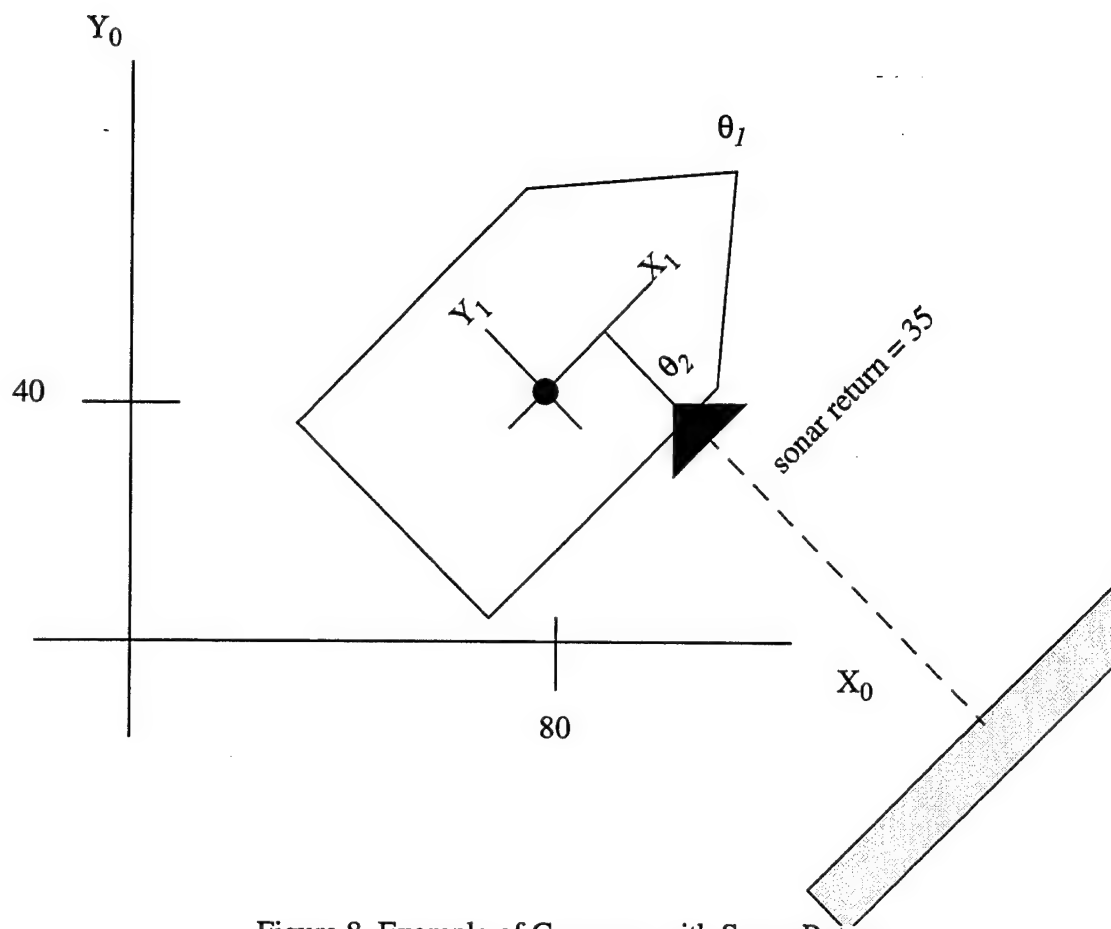


Figure 8. Example of Compose with Sonar Return

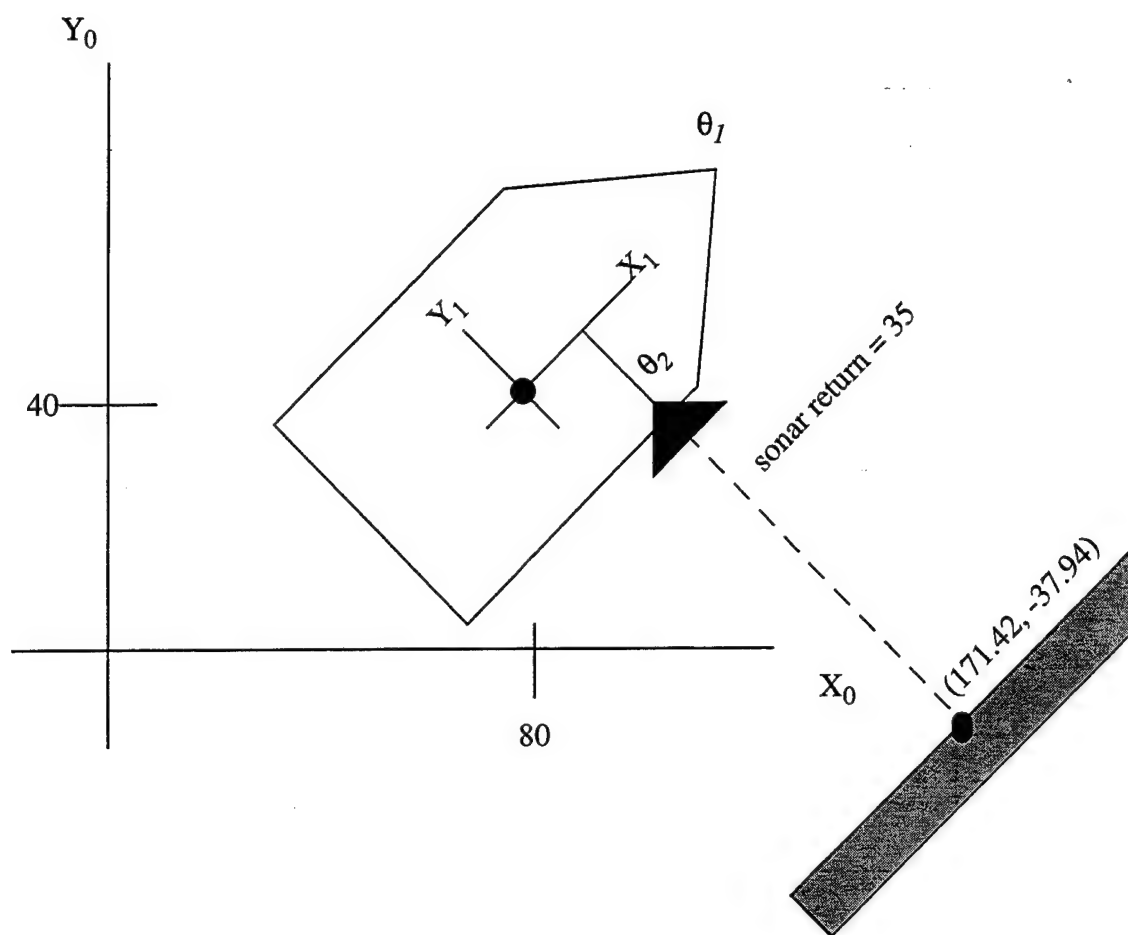


Figure 9. Result of Compose on Example

B. SONAR FUNCTIONS

Sonar functions are found in sonarcad.c, sonarmath.c, sonario.c, sonarsys.c, and sonarlog.c, which are part of Yamabico's MML, the name for the entire set of code for Yamabico. The following are those functions which are available for use in the user.c and a brief description.

1. Enable Sonar

Syntax: void enable_linear_fitting(n)

int n;

Description:

The user calls this function passing in the sonar that is to be enabled. On Yamabico there are 12 available sonars. Each sonar should be enabled individually.

2. Disable Sonar

Syntax: void disable_sonar(n)

int n;

Description:

The user calls this function passing in the sonar that is to be disabled. On Yamabico there are 12 available sonars. Each sonar should be enabled individually.

3. Get Sonar Returns

Syntax: double sonar(n)

int n;

Description:

The user calls this function and passes in the sonar number that range data is wanted from. If no echo is received, then an INFINITY(1.0e6) is returned. If the distance is less than 10 cm, then a 0 is returned. If the sonar return is between 9 cm to 409 cm, then that floating point number will be returned in centimeters.

4. Get Global Sonar Returns

Syntax: `posit global(n)`

`int n;`

Description:

The user calls this function and passes in the sonar number that global range data is wanted from. The function will return a structure of type `posit`, which contains `gx` and `gy`, the global x and y coordinates.

5. Enable Linear Fitting

Syntax: `void enable_linear_fitting(n)`

`int n;`

Description:

The user calls this function and passes in the sonar number, so that linear fitting is applied to sonar returns. This will enable the robot to determine whether sonar returns are walls, or some type of linear surface.

6. Disable Linear Fitting

Syntax: `void disable_linear_fitting(n)`

`int n;`

Description:

The user calls this function and passes in the sonar that linear fitting is to be disabled on.

7. Set Parameters In Linear Square Fitting

Syntax: `void set_sonar_parameters(c1, c2)`

`float c1,c2;`

Description:

Allows the user to adjust constants which control the linear fitting algorithm. C1 is a multiplier to allow more leniency for greater sonar ranges, and C2 will adjust the

tolerance allowed for sonar ranges being off the linear line being collected. Both are used to determine if an individual data point is usable for the algorithm. The default values are initialized to 0.02 and 5.0 respectively. For more information on C1 and C2 refer to Chapter V.C of this thesis.

8. Enable Data Logging

Syntax:

```
void enable_data_logging(n,filetype,filename)
```

```
int n,filetype,filename;
```

Description:

The user calls this function and passes in the sonar, the type of file data to be collected, and which file array (0, 1, 2, or 3) to collect the data in. There are three types of file data that can be collected. The first is raw data, the second is global data, and the third is segment data.

9. Disable Data Logging

Syntax: void disable_data_logging(n,filetype)

```
int n, filetype
```

Description:

The user calls this function and passes in the sonar, the type of file data to be collected, and which file array (0, 1, 2, or 3). The type of file data that is to cease being collected is designated, either raw data, global data, or segment data.

10. Set Logging Interval

Syntax: void set_log_interval(n,d)

```
int n, d;
```

Description:

The user calls this function passing an integer designating how often the sonar data being collected should be written to the file collecting the data. The default value is 13, which for a speed of 30 centimeters per second and sonar sampling time of 25 milliseconds.

would record a data point approximately every 10 cm. To collect all sonar data you pass in 1, so that every sonar return is recorded.

11. Transfer Raw Data To Host

Syntax: void xfer_raw_to_host(filename, filename)

int filename, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the raw sonar data collected.

12. Transfer Global Data To Host

Syntax: void xfer_global_to_host(filename, filename)

int filename, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the global sonar data collected.

13. Transfer Segment Data To Host

Syntax: void xfer_segment_to_host(filename, filename)

int filename, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the segment sonar data collected

C. DATA LOGGING PROCEDURE

After *Yamabico* has completed its mission, recorded sonar data can be downloaded and checked to ensure that the hardware is performing optimally. The data that can be logged includes global sonar data, raw sonar data, segment sonar data, and the motion trace data of the robot. Once the robot has stopped, the data designated to be logged in user.c can now be downloaded. A message on the powerbook will instruct the user to connect the

phone cable to the robot. Once the phone line is connected, the user must hit the space bar, then the character g, and the space bar once more. The data will then be downloaded to the workstation. Once the download is completed, a bell sound will be heard from the powerbook on the robot. This is required for each type of data being logged.

V. SONAR CHARACTERISTICS EXPERIMENT RESULTS

To successfully use sonars in motion planning and obstacle avoidance, it is necessary to understand what sonar data you can expect in distinct rectilinear configurations. This allows determination of which cases will successfully avoid obstacles, and which cases will be unable to determine a safe path given only input from the sonars.

Several motion planning experiments were conducted in Spanagel Hall at the Naval Postgraduate School. During all cases the left and right sonars (#5, #7) were enabled. This provided complete scans of the world in which the robot was moving. The Linear Fitting Algorithm constructs segments from individual sonar returns.

The Linear Fitting Algorithm has 5 cm allowance. This means, a segment is continually generated until the new return's distance (image) to the already formed segment exceeds the 5 cm allowance of the algorithm (Figure 10.).

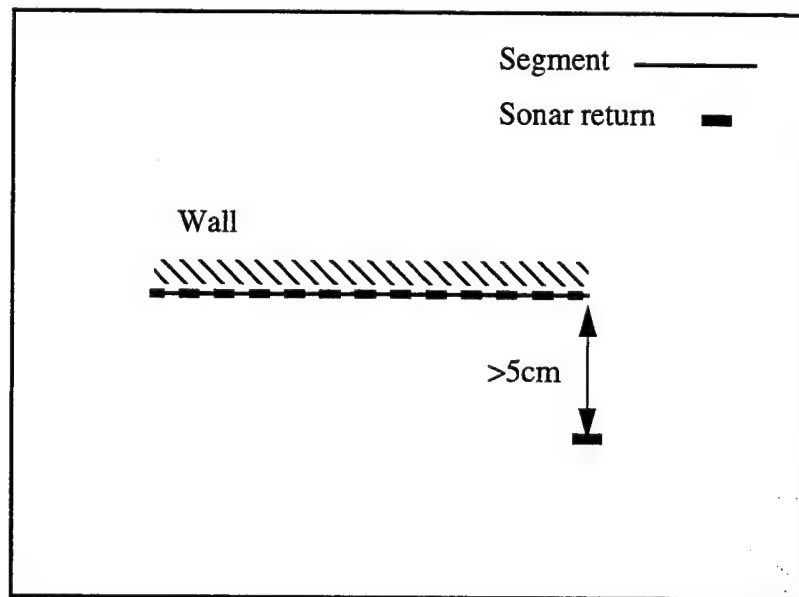


Figure 10. Allowance in Linear Fitting Algorithm

A. CASE 1

The robot moves using its left and right sonars in a translational scan as in Figure 11. We expect to receive accurate data to recognize the wall. The Linear Fitting Algorithm recognizes two walls. More importantly, the locations of the doorway cavities are correctly determined.

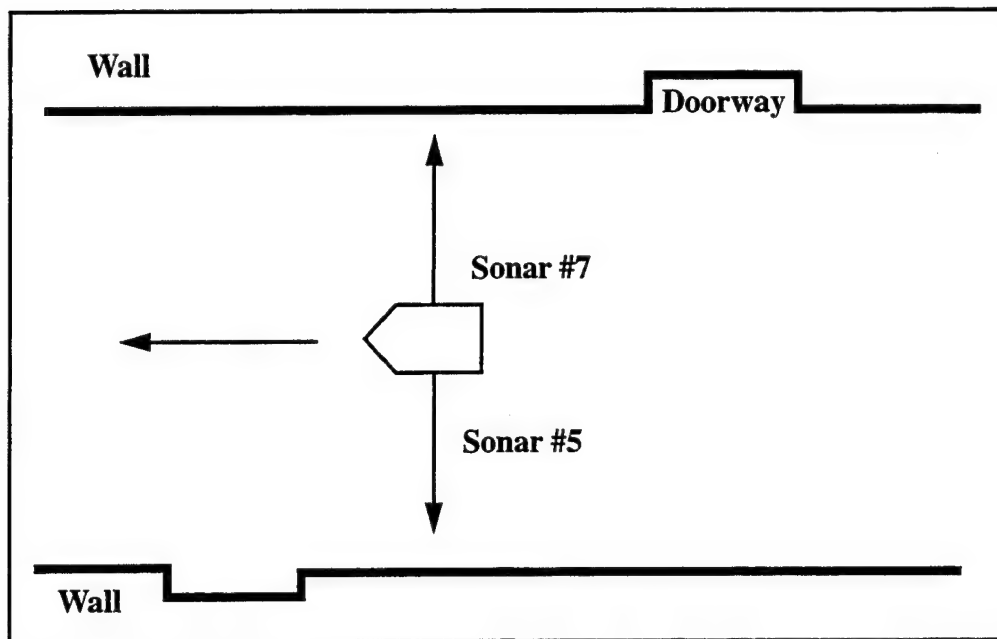


Figure 11. Case 1

Thus, in case 1 the robot's understanding of the world is precise and the result exhibits good linearity. Figure 12. shows the experiment results.

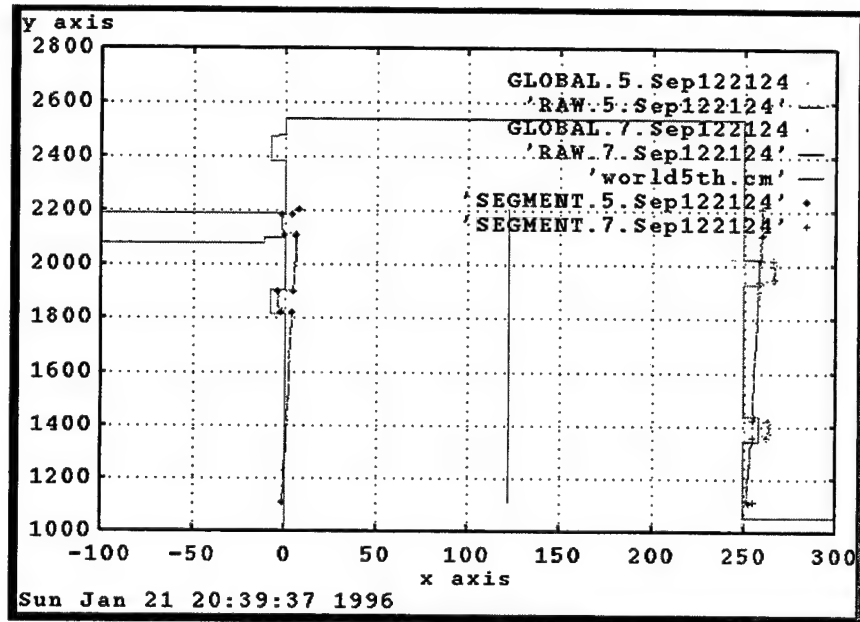


Figure 12. Case 1-Experiment Results

B. CASE 2

Case 2 is designed to observe the behavior of the "EnableLinearFitting" function, as depicted in Figure 13. The robot is in a translational scan of a wall which ends in a corner. We expect results that will not accurately depict the corner.

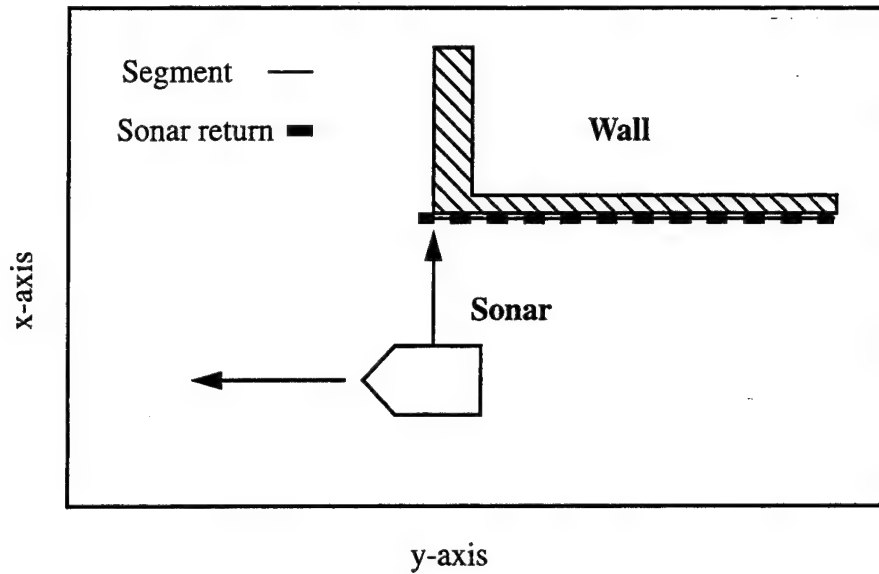


Figure 13. Case 2

The robot moves parallel to the wall. Prior to reaching the corner, many sonar returns contributed to the segment. A sonar return immediately after the robot passed the corner was added to the segment. This is within the 5 cm allowance. However, the accuracy of the generated line segment is not significantly affected. The experiment results is shown in Figure 14.

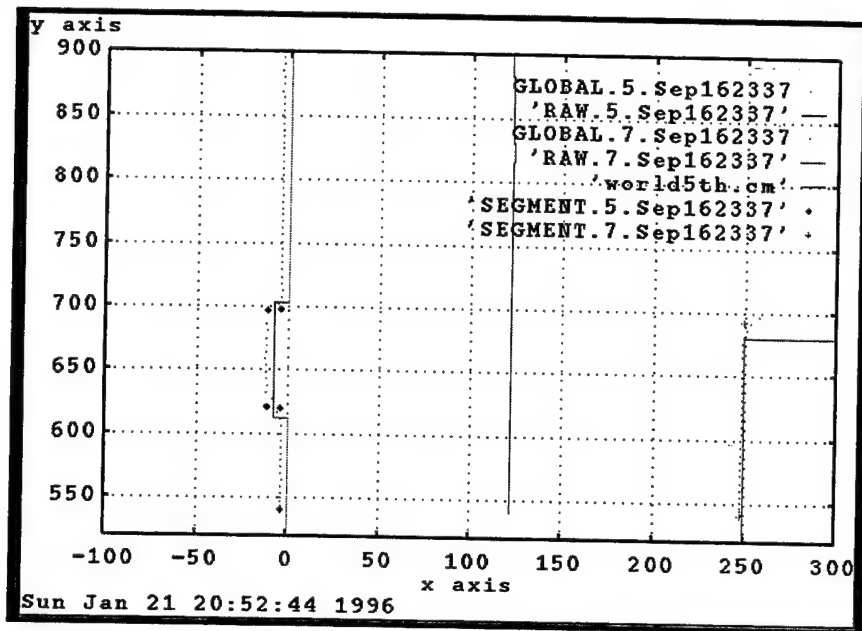


Figure 14. Case 2-Experiment Results

C. CASE 3

The robot uses its right sonar in a translational scan of a corner in a situation shown in Figure 15. The expected results may not accurately depict the corner due to the low amount of reflection which distorts the representation of the angle of the wall.

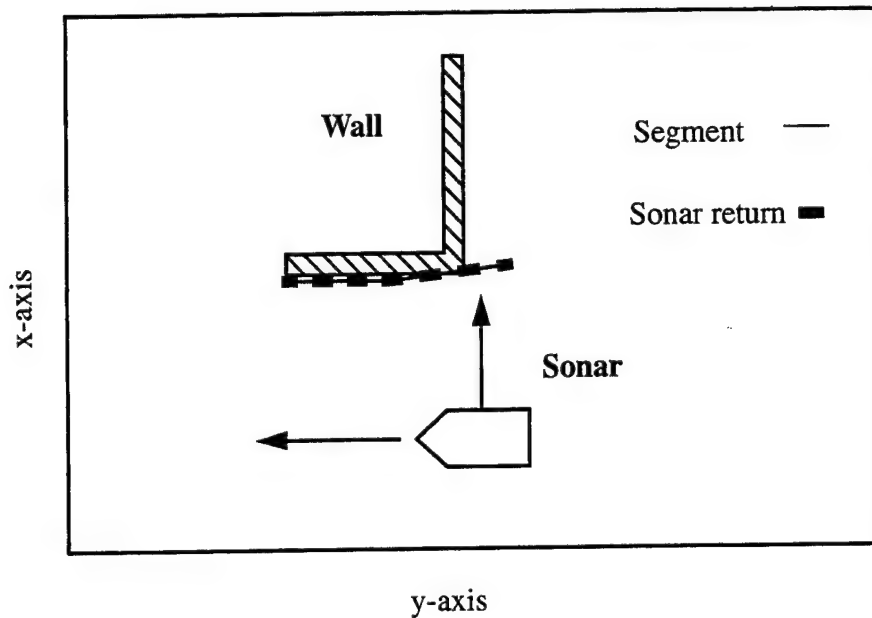


Figure 15. Case 3

The actual results showed the expected angular distortion. Some sonar returns from the wall are received before the robot's y-coordinate reaches the y-coordinate of the corner. Next, as the robot move parallel to the wall we get more sonar returns. These returns along with the prior ones, generate a first segment.

Then, another segment is generated right after the Linear Fitting Algorithm's allowance prohibits the first segment to continue. The two segments together give us a bended representation of the wall. The experiment results is shown in Figure 16.

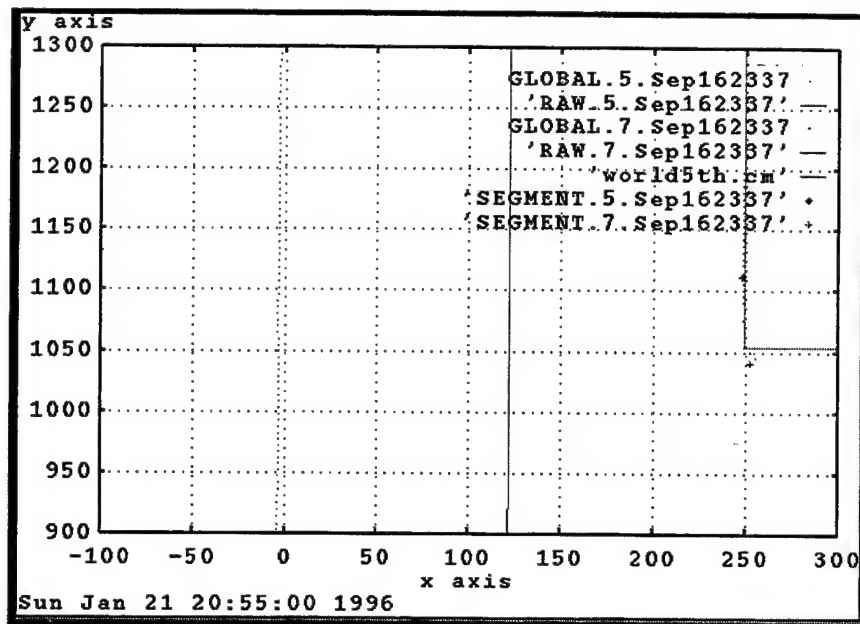


Figure 16. Case 3-Experiment Results

VI. THEORY OF POLYGONS

This chapter presents definitions used in the development of the theory and includes the concepts of polygons, subpolygons, and rectilinear worlds. Basic terminology and definitions, which form the basis of the discussions to follow, are now covered [KAN95a] [KAN95c].

A. DISTANCE AND BISECTORS

We assume a global two-dimensional Cartesian coordinate system in a plane. Distances will be measured as Euclidean distance. The distance $d(p, q)$ between two points $p = (x_p, y_p)$ and $q = (x_q, y_q)$ is defined by the usual L_2 norm:

$$d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2} \quad (\text{Eq. 27})$$

Assume there are $n (n > 2)$ points in a plane that make up the world, W .

$$W = \{p_1, \dots, p_n\} \quad (\text{Eq. 28})$$

The bisector of two points is a straight line which bisects and is perpendicular to a line connecting the two points. A bisector $bs(p_i, p_j)$ of points p_i and p_j with $1 \leq i, j \leq n, i \neq j$ is defined as

$$bs(p_i, p_j) = \{p \mid d(p, p_i) = d(p, p_j)\} \quad (\text{Eq. 29})$$

Bisectors play important roles in this theory in several ways. Obviously, $bs(p_i, p_j)$ is a line for every pair (p_i, p_j) of points.

Voronoi regions are those regions in which any point in the region is closer to an obstacle than any other obstacle. As is well-known, the Voronoi region $V(p_i)$ of a point p_i with $1 \leq i \leq n$ is defined as

$$V(p_i) = \{p \mid (\forall j) d(p, p_i) \leq d(p, p_j)\} \quad (\text{Eq. 30})$$

B. POLYGONAL WORLDS

Throughout this discussion we will use the rectilinear world illustrated in Figure 17.

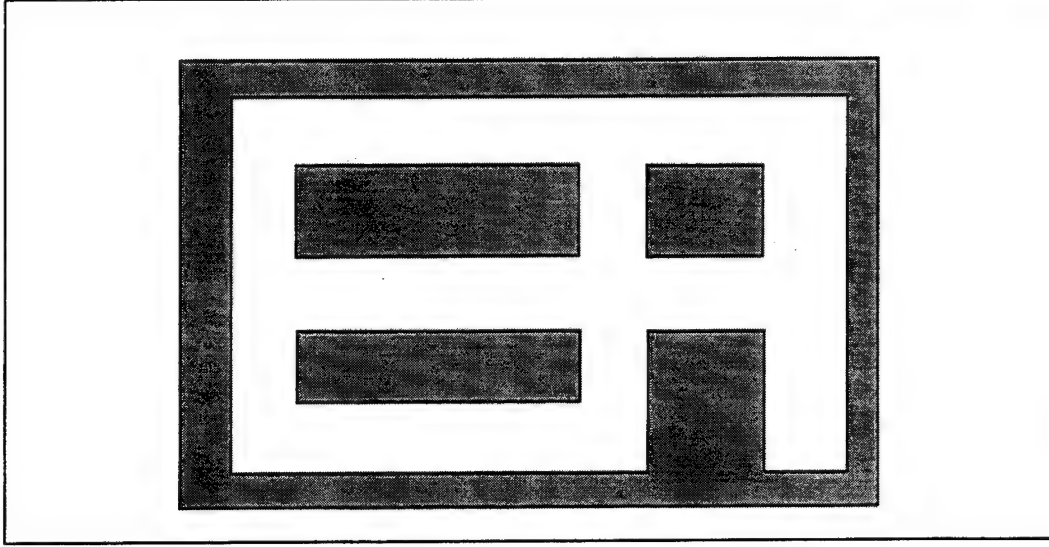


Figure 17. Example World

as a concrete example of the concepts developed herein. Note that the inner white regions indicate areas in free space while the darker areas indicate the rectilinear obstacles.

1. Polygons

Let a **polygon**, Γ , be defined by an ordered circular list of vertices, located in a two dimensional plane, \mathcal{R}^2 , satisfying the following conditions:

- i) any three consecutive points in the sequence are not colinear,
- ii) two distinct edges (v_j, v_{j+1}) and (v_i, v_{i+1}) do not have intersections except in the case that they share a common end point.

Therefore, polygon, Γ , is defined as:

$$\Gamma = (v_1, \dots, v_l), l \geq 3 \quad (\text{Eq. 31})$$

where Γ consists of the set of points which are either on the boundary of Γ or in the interior of Γ . The **boundary** of the polygon Γ , denoted $\partial\Gamma$, consists of those points of a straight edge connecting the vertices in order. The last vertex, v_l , is connected to the first vertex to close the directed loop. In this theory, free space will always exist to the right of the directed boundary loop. The **interior** of Γ , denoted $Int(\Gamma)$, is defined as the set of points to the left of the boundary. The **exterior** of Γ , denoted $Free(\Gamma)$, is defined as the set of points to the right of the boundary and can therefore be defined as $\mathcal{R}^2 - \Gamma$. A **normal polygon** is a polygon whose ordered list of vertices produces a counterclockwise boundary loop. See Figure 18. Normal polygons will therefore represent obstacles inside the boundaries of

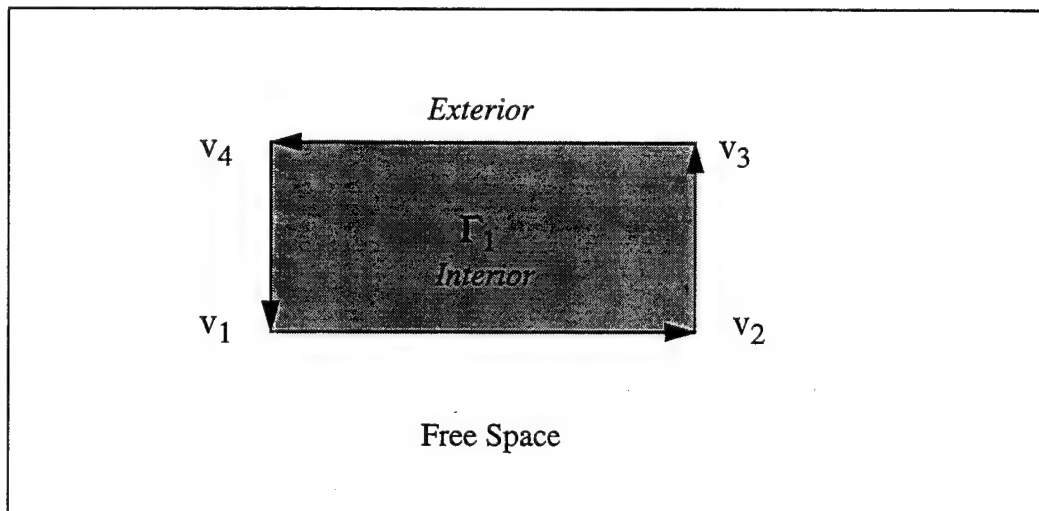


Figure 18. Normal Polygon

the world. An **inverted polygon** is a polygon whose ordered list of vertices produces a clockwise directed boundary loop. See Figure 19. An inverted polygon serves as the outer boundary of the world. Recall that an assumption used throughout is that the world is bounded by an outer inverted polygon. A **rectilinear polygon** is a polygon in which every edge is parallel to either the global x or y orthogonal axes.

Definitions of concave and convex with regard to vertices and polygons are covered here. The exterior angle will serve as a measure to determine whether a vertex is concave

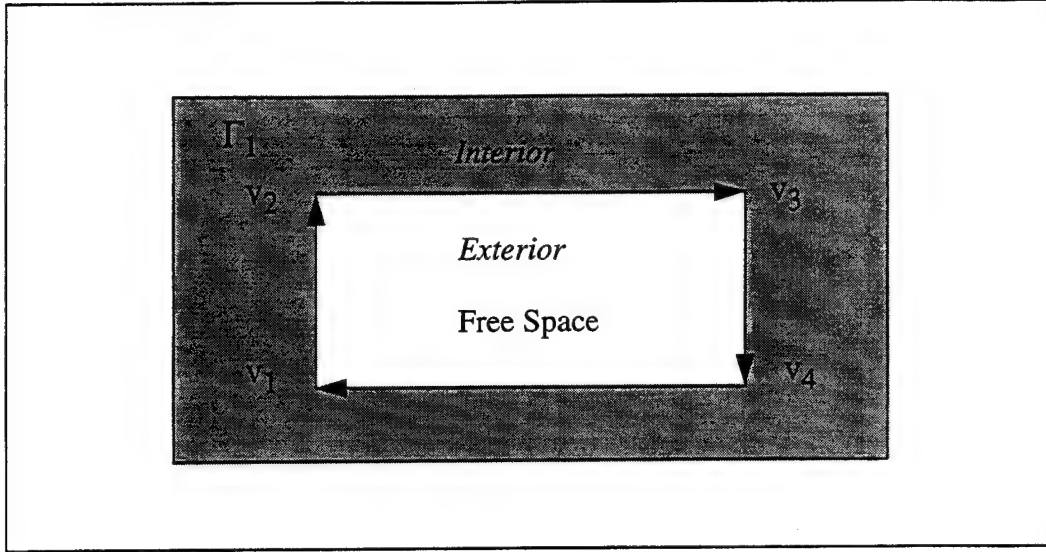


Figure 19. Inverted Polygon

or convex. (See 20. .) Let $\psi(v_i, v_{i+1})$ represent the direction in global coordinates from v_i

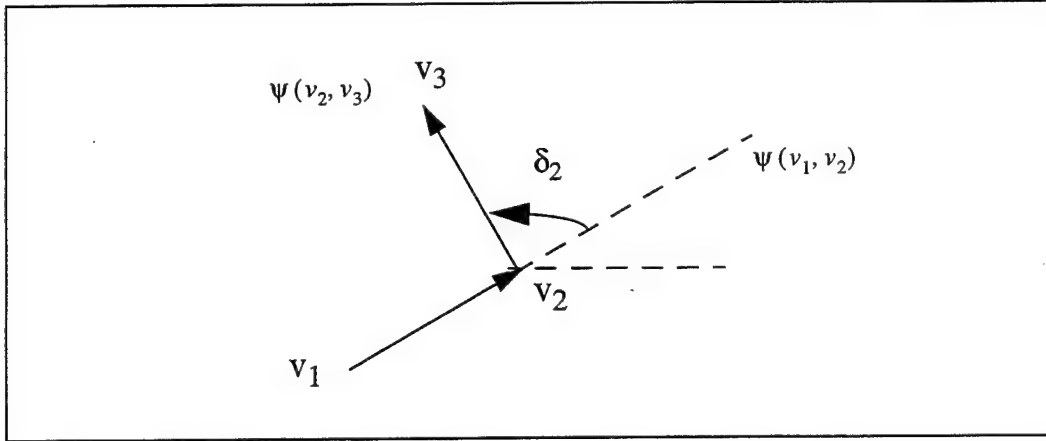


Figure 20. Exterior Angle of a Convex Vertex

to v_{i+1} . The exterior angle, δ_i , induced at the i th vertex, v_i , is defined as:

$$\delta_i = \text{normalize}(\psi(v_i, v_{i+1}) - \psi(v_{i-1}, v_i)) \quad (\text{Eq. 32})$$

Note that the difference between the directions is *normalized* to fall within $(-\pi, \pi]$. A vertex, v_i , is said to be a **convex vertex** if $\delta_i > 0$. Otherwise, the vertex is said to be a **concave vertex**. A polygon is said to be a **convex polygon** if all of its vertices are convex,

otherwise it is a **concave polygon**. For more details on the properties of polygons see [30]. Several examples are presented in the Figures below. Figure 21. shows examples of convex polygons, while Figure 22. shows examples of concave polygons.

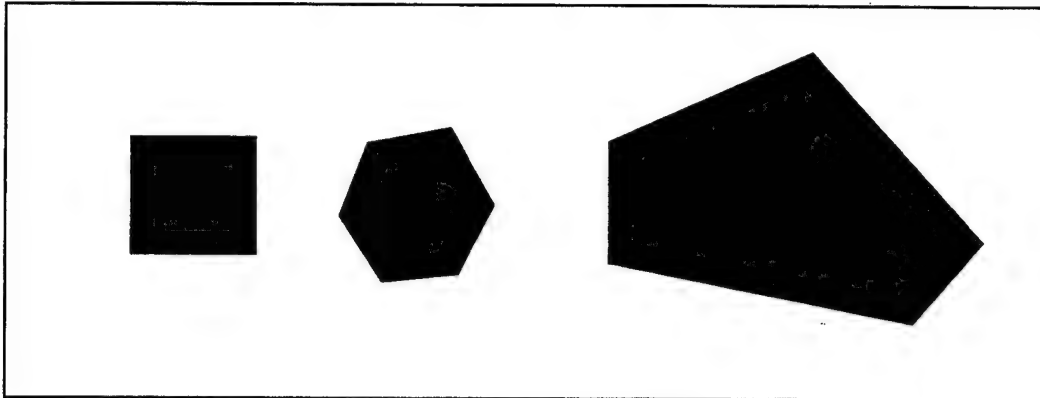


Figure 21. Convex Polygons

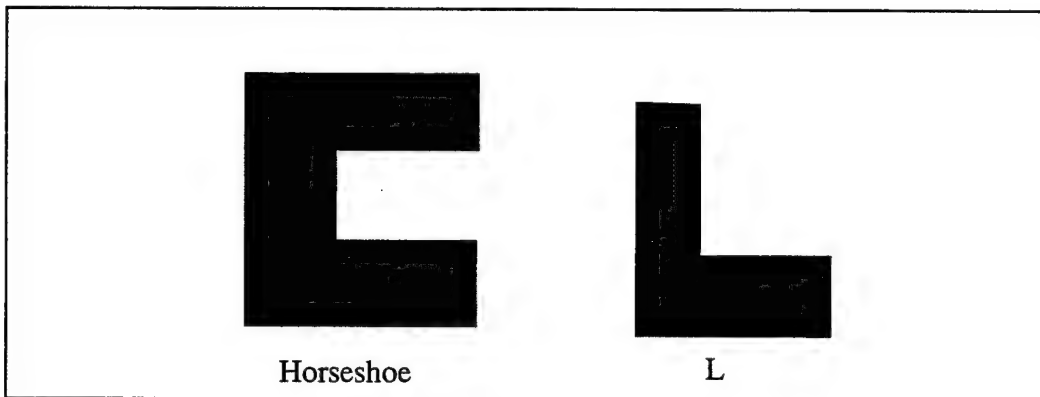


Figure 22. Concave Polygons

2. Subpolygons

Let, $\Gamma = (v_1, \dots, v_l), l \geq 3$ be a polygon. It is desired to decompose Γ into smaller pieces, called subpolygons. At this point the decomposition is dealing with decomposing polygons into subpolygons and is not a decomposition of free space. It is, however, a step in the future process of free space decomposition.

If the polygon Γ is convex, i.e., if all of the vertices are convex, we stipulate that the polygon Γ itself is a unique subpolygon in Γ . If Γ is concave, i.e., if there is at least one concave vertex in Γ , the polygon can be broken up into one or more subpolygons. In

that case, the first vertex in the subsequence of vertices defining a subpolygon is set as one of the concave vertices. The subsequence continues until it encounters another concave vertex, which becomes the last vertex in the subpolygon's defining subsequence. A subsequence

$$\gamma = (v_p, v_{t+1}, \dots, v_u), t \leq u \quad (\text{Eq. 33})$$

of Γ is said to be a **subpolygon** of Γ , if v_t and v_u are concave and if all the vertices v_{t+1}, \dots, v_{u-1} are convex. v_t and v_u are said to be the *end-vertices* of the subpolygon γ .

In a special case where there is only one concave vertex v_1 in Γ ,

$$(v_1, \dots, v_p, v_1) \quad (\text{Eq. 34})$$

is the unique subpolygon.

Figure 23. shows the decomposition of the polygons in the example world

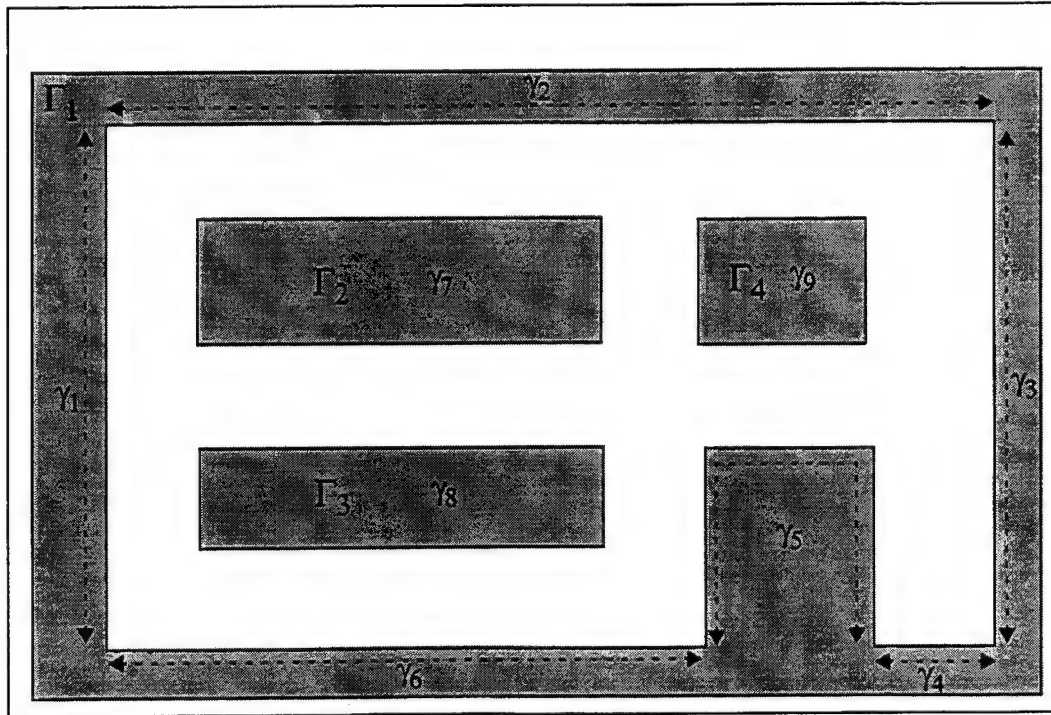


Figure 23. A Rectilinear Polygonal World

into subpolygons. Note that Γ_1 , which is a concave polygon, consists of six subpolygons.

The encompassing shape of each subpolygon that makes up Γ_1 is shown using a dotted line with arrowhead ends. The other convex polygons consist of only one subpolygon each. Additional examples of polygonal decomposition into subpolygons are given in Figure 24.

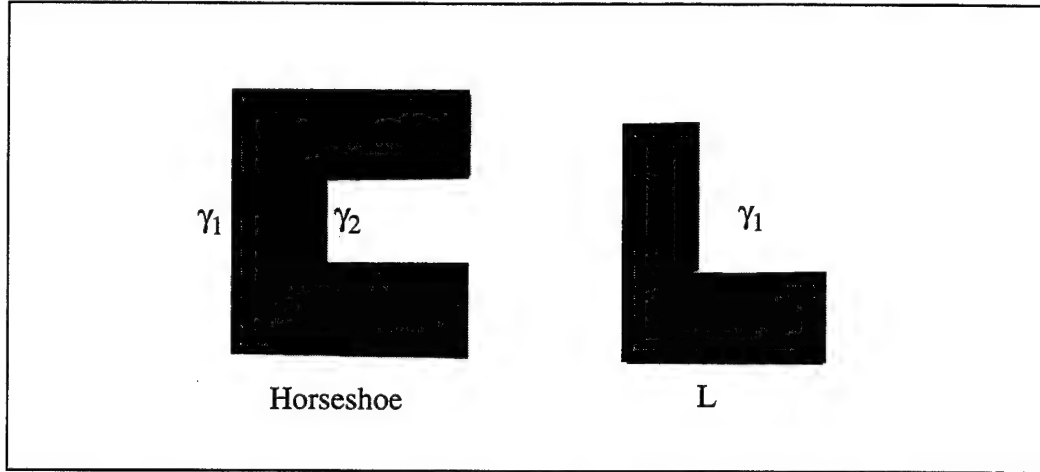


Figure 24. Subpolygon Decomposition of Concave Polygons

The polygon on the left of the figure consists of two subpolygons. The polygon on the right of the figure consists of only one subpolygon.

Lemma 1: Any polygon Γ is uniquely divided into a finite number of subpolygons

$$(\gamma_1, \gamma_2, \dots, \gamma_n) \quad (\text{Eq. 35})$$

with keeping the order of occurrences of vertices in Γ . Each convex vertex v in Γ belongs to one and only one subpolygon.

If two subpolygons γ and γ share the same end-vertex, they are said to be **adjacent** and we write $\text{adj}(\gamma, \gamma)$. For example, in Figure 23., γ_1 is adjacent to γ_2 . Also, γ_5 is adjacent to γ_6 . However, γ_1 is not adjacent to γ_5 .

3. Distance from a Point to a Subpolygon

In a polygonal world W , let p be a point in its free space and γ one of the subpolygons in W respectively. Then $d(p, \gamma)$ means the minimal visible distance from p to γ . If p and γ are not visible to each other in this world, $d(p, \gamma) = \infty$.

The **image**, $im(p, \gamma_i)$, of the point p on the subpolygon γ_i is defined as the closest point on γ_i from p . The image may lie on a convex vertex or within an edge of a subpolygon. The $im(p, \gamma_i)$ will have as components the global x and y coordinates, denoted $im(p, \gamma_i) \equiv (x_{im(p, \gamma_i)}, y_{im(p, \gamma_i)})$

Proposition 1: For any point p and a subpolygon γ in a world, if γ has no vertex in the interior of the convex hull of Γ which contains γ , then the image of p on γ is unique.

Proof: Since there are no vertices of γ in the interior of the convex hull of Γ , then by definition of a convex hull, the vertices of γ must lie on the boundary of the convex hull. Since the convex hull is a convex polygon, then there is only one image point to γ .

4. Polygonal World

Assume a world W is given which consists of a finite number of polygons and each polygon is divided into one or more subpolygons. Assume there are n non-overlapping polygons in the world.

$$W = \{\Gamma_1, \dots, \Gamma_n\}, n \geq 1 \quad (\text{Eq. 36})$$

$$W = \bigcup_i \Gamma_i \quad (\text{Eq. 37})$$

$$Int(W) = \bigcup_i Int(\Gamma_i) \quad (\text{Eq. 38})$$

$$Free(W) = \bigcup_i Free(\Gamma_i) = \mathbb{R}^2 - W \quad (\text{Eq. 39})$$

$Free(W)$ is called **free space** of W . ∂W is defined as the boundary of W . $Int(W)$ is defined as the interior of W . A **rectilinear world** is a world in which every polygon contained in the world is rectilinear.

VII. GLOBAL MOTION PLANNING / CONVEX

DECOMPOSITION

A. PATH CLASS

A world $W = \{B_0, B_1, \dots, B_n\}$, $n \geq 1$, consists of n simple polygons (holes) and another simple polygon B_0 which defines the outmost boundary [KAN95b]. The *free space* of W is the inside of B_0 minus the union of the other n polygons' inside and is denoted by $\text{Free}(W)$. (Although polygons and borders can be non rectilinear in this theory, all examples given are orthogonal for simplicity.)

A *path* in W is a continuous function $f: [0,1] \rightarrow \text{Free}(W)$. The two points $f(0)$ and $f(1)$ are called its *endpoints*. If they are distinct, we usually denote $f(0)$ as a start S and $f(1)$ goal G . We assume that f is rectifiable (its length is finite). Two paths f and f' with the same endpoints are said to be homotopic if f can be continuously transformed into f' without moving both endpoints and without running over any polygons. If f and f' are homotopic, we write $f \equiv f'$. In Figure 25., $f_1 \equiv f_2$ and $f_3 \equiv f_4$. The relation \equiv is an

equivalence relation and defines equivalence classes of paths that share the same endpoints. These equivalence classes are called path classes.

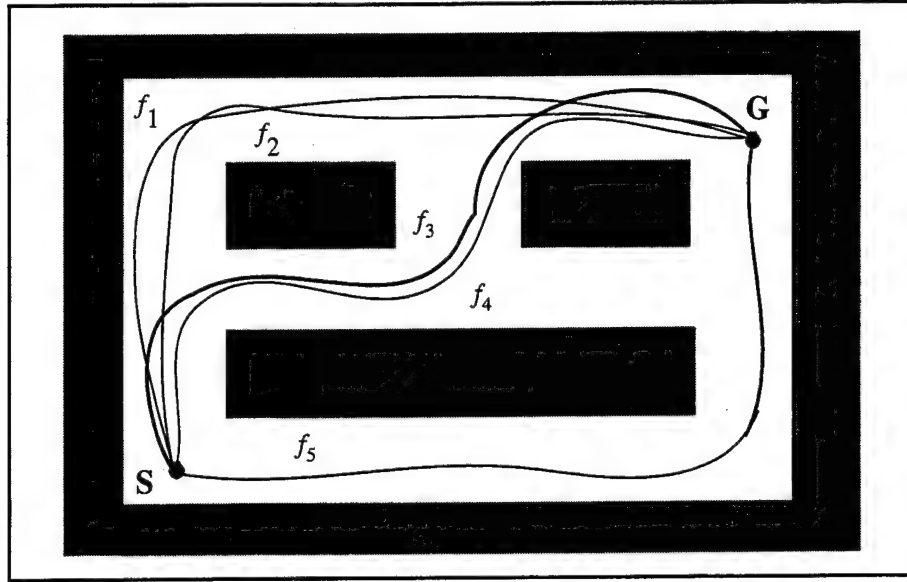


Figure 25. Paths

The motion planning problem is a problem of finding the “optimal” motion for a robot given a world W and a pair of configurations (position and direction). We divide this problem into two: (i) finding the “optimal” path classes (*the global motion planning problem*) and (ii) finding the “optimal” motion in the given optimal path class (*local motion planning problem*). This thesis mainly deals with the first problem. It is known that the number of path classes goes faster than any polynomial function of n , where n is the number of holes in the world. Therefore, we must find an efficient algorithm for robot motion planning.

B. DECOMPOSITION

The boundary ∂W of a world is defined as the union of all polygon boundaries in W . A border in a world is a straight line segment L , (i) its both endpoints are on the boundary ∂W of the world, and (ii) the open segment L is a subset of the free side $\text{Free}(w)$

of the world. A finite set D of borders in W is called a *decomposition* of the world, if D satisfies the condition that if any two borders share a point, it is one of the endpoints for each border. Figure 26. shows two ways of decomposing the world shown in Figure 25.

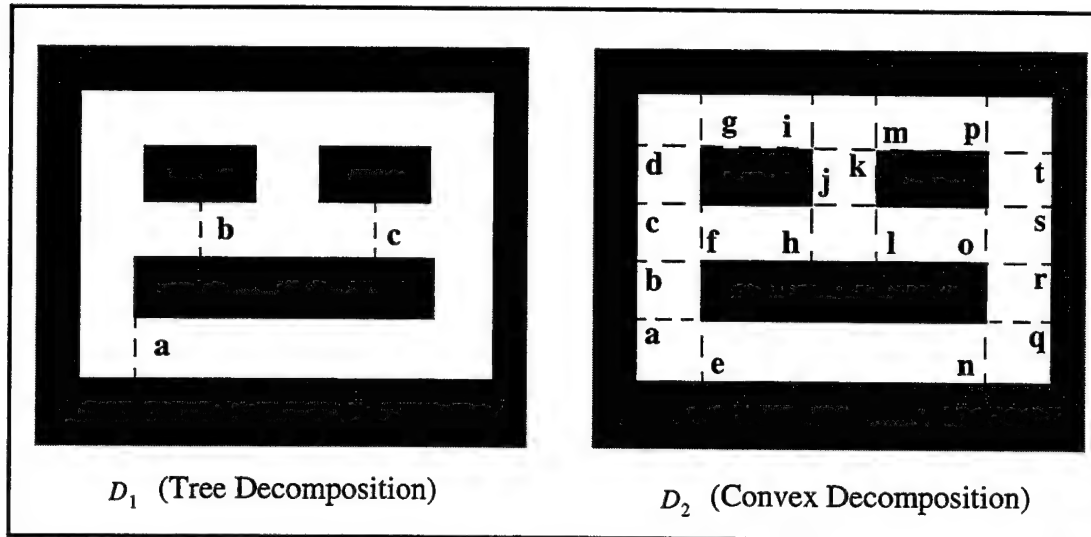


Figure 26. Decompositions

When we say a path f intersects a border L , we also specify the orientation how f intersects it. By adding an orientation to L , we say f *positively-intersects* or *negatively-intersects* L depending on the direction of f . An *oriented border* is represented by L^+ or L^- . (Although the way an orientation is defined upon a border is arbitrary, we follow this convention: We give an orientation to each border so that if a path intersects a horizontal border upward or it intersects a vertical border rightward, we say the path positively-intersects the border (L^+), otherwise, negatively-intersects (L^-).)

Given a path f in a decomposed world, we define its *border sequence* $\lambda(f)$, as an ordered list of oriented borders which f intersects. For instance, in Figure 27., the border sequence of the two paths f_1 and f_2 are

$$\begin{aligned}\lambda(f_1) &= \langle L_1 + L_6 + \rangle \\ \lambda(f_2) &= \langle L_2 + L_4 + L_5 + \rangle\end{aligned}$$

In Figure 27., the positive orientation of each border is shown by an arrow. If either of the endpoints is on a border, this oriented border is also included in its border sequence. If f does not intersect any borders and if both endpoints of f are not on any border, $\lambda(f)$ is defined as the empty sequence ϵ . If f stays on one border, $\lambda(f) = \epsilon$.

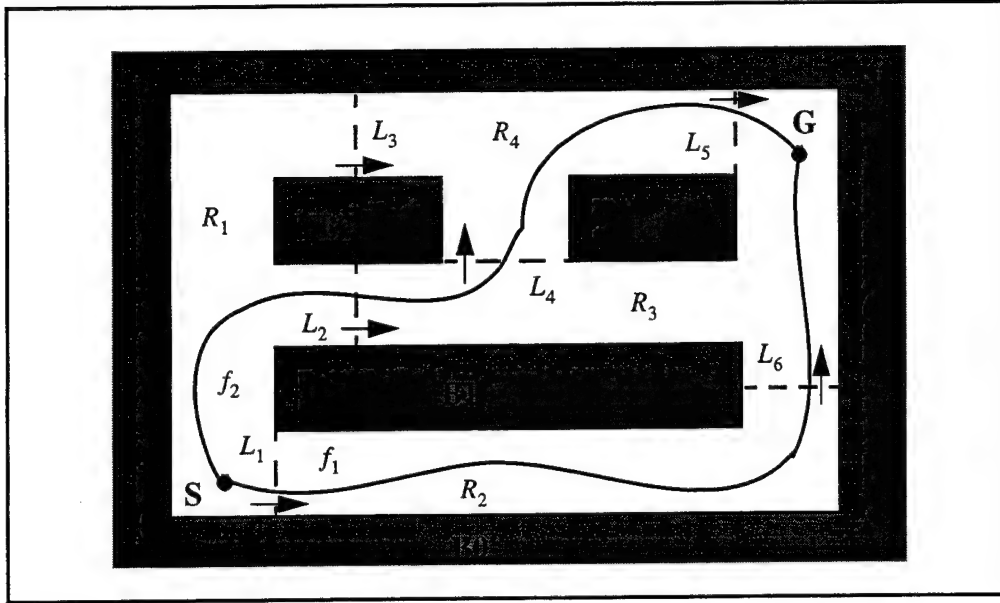


Figure 27. Paths in a Decomposed World

The border sequence of a path f in Figure 28. is

$$\lambda(f) = \langle L_2 + L_2 - L_1 + L_6 + \rangle$$

This kind of a path can be simplified, because this path is homotopic to the path f_1 in Figure 27. Another example of an unreasonable path is one which has infinitely many intersections with D . Through the following definition and lemma, we can avoid dealing with these pathological paths. A path f in a decomposed world is said to be *regular* if it

divide the free space into more than one region, this decomposition is called a *tree decomposition*.

Proposition 1 *A homotopic decomposition of a given world with the minimum number of borders is a tree decomposition. The number of borders in a tree decomposition and the number of holes in the world are the same. For an arbitrary world, there exists at least one tree decomposition.*

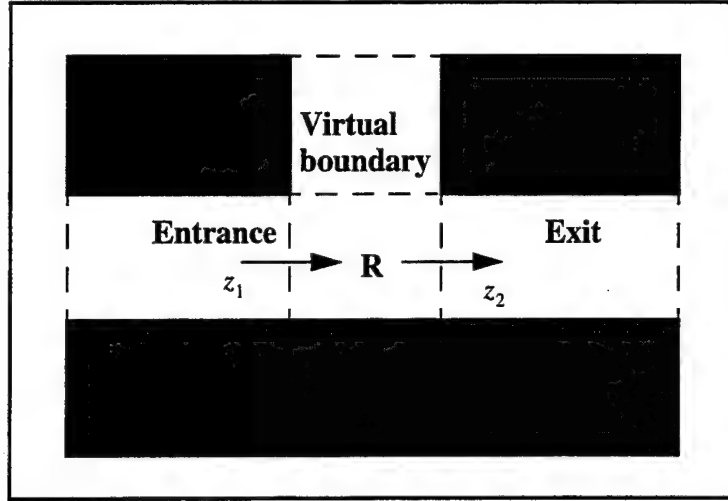


Figure 29. Oriented Region

Suppose $z_1 z_2$ is a subsequence of the border sequence $\lambda(f)$ of a path f , where z_1 and z_2 are oriented borders. Then there is a region R such that f enters into R through the oriented border z_1 and exits R through the oriented border z_2 . We call the borders z_1 and z_2 the *entrance* and the *exit* of the region R in relation to the path f . Now we define an *oriented region* from the region R by artificially closing all borders belonging to R except for the entrance and exit. The borders which are artificially closed are called *virtual boundaries* of the oriented region. Thus, an oriented region is bounded by (i) entrance, (ii) exit, (iii) portions of the world boundary, and (iv) virtual boundaries. The union of the parts (iii) + (iv) is divided into two: the *left boundary* and *right boundary* of the oriented region.

A left or right boundary of an oriented region may be one point, but cannot be empty (Figure 29.).

The initial region of a path f is the region where S belongs. If S is on a border, the initial region is empty. The final region of a path is the region where G belongs. If G is on a border, the final region is empty. (The left/right boundaries of the first or last regions are not defined, since they have the exit or the entrance only, not both.)

The *path region* of a path is the union of the first region, all the oriented regions, and the last region of the path. The left (right resp.) boundary of a path region is the union of the left (right resp.) boundaries of all the oriented regions. A path region is also contractible. The path region of the path f_2 in Figure 27. in a decomposed world by D_2 as shown in Figure 26. becomes the region shown in Figure 30. (If a path region is constructed on a tree decomposed world, it becomes extremely complex. This is one reason why convex decompositions are more appropriate in practical work).

A path region is a geometric representation of path classes. Therefore, this concept of *path region* is essential in providing the main result which follows. This concept is also extremely useful in the local motion planning problem, although this thesis does not address this topic.

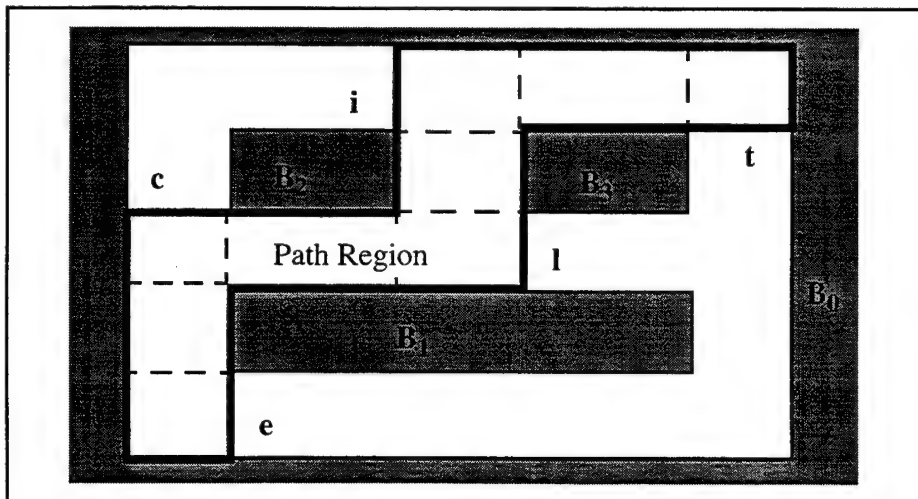


Figure 30. Path Region

In the following proposition we use the border sequence as a symbolic representation of path classes.

Proposition 2 (Main Result) *Suppose a homotopically decomposed world W and two endpoints in $Free(W)$ are given. For an arbitrary pair of regular paths f and f' connecting these endpoints, $f \equiv f'$ if and only if $\lambda(f) = \lambda(f')$.*

A sketch of the proof: (I) Only-if part: Let us consider the path region of f . Since f and f' are both regular, f' is also confined in that path region.

(II) If part: The border sequence $\lambda(f)$ defines a path region. Since the path region is contractible with its left / right boundaries, $f \equiv f'$ follows.

D. CONVEX DECOMPOSITION

The connectivity graph method described in the previous Section works for any homotopic decomposition. However, the use of convex decompositions generally gives better results, where a decomposition D is said to be a *convex decomposition* if each region generated by D is convex. The reasons are (i) convex decompositions makes cost-evaluation for basic connectivity graph easier, and (ii) convex decompositions makes the local motion planning task simpler.

Proposition 3 *Every convex decomposition is a homotopic decomposition.*

Proposition 4 *A vertical decomposition or horizontal decomposition is a convex decomposition.*

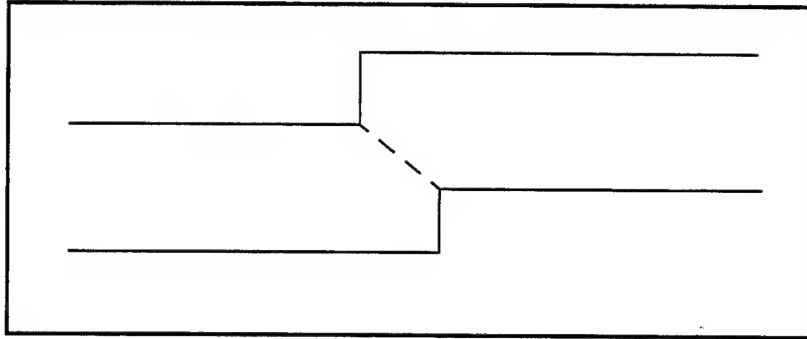


Figure 31. Non-Orthogonal Border

In the world shown in Figure 31., the diagonal border is better than any vertical or horizontal ones from several aspects. This observation supports the advantage of convex decompositions over vertical or horizontal decompositions.

VIII. DIJKSTRA'S ALGORITHM

A. CONNECTIVITY GRAPH OF THE WORLD

The *global motion planning* problem is the problem of finding the “optimal” path class to connect given *start* and *goal* configurations [KAN95a]. The homotopic decomposition method is an extremely useful tool to solve this problem. The *connectivity graph* is defined for a homotopically decomposed world.

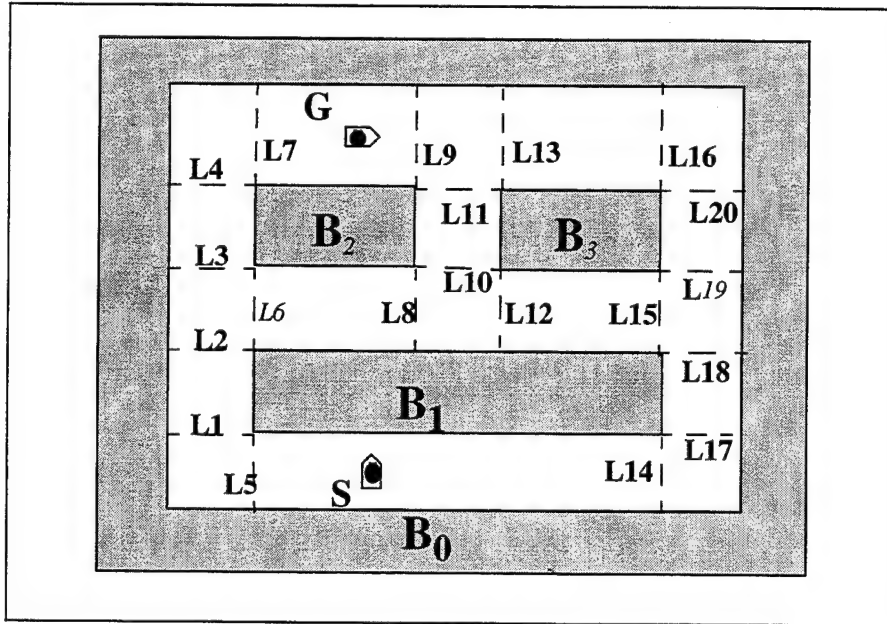


Figure 32. World Decomposition

A node is assigned for each border in the convexly decomposed world. When two borders, L_1 and L_2 , belong to the same region, an edge is created. A cost for this edge is defined as the energy (or time) for the vehicle to make a motion from L_1 to L_2 or from L_2 to L_1 . Therefore, these edges are undirected. This cost not only reflect the distance but the turn needed to make the motion. It may also reflect the safety in the region (i.e. if the region is narrow, the cost is high). Since the region is convex, any two borders in a region are

visible and the cost evaluation is relatively straightforward. This graph is called a *basic connectivity graph*.

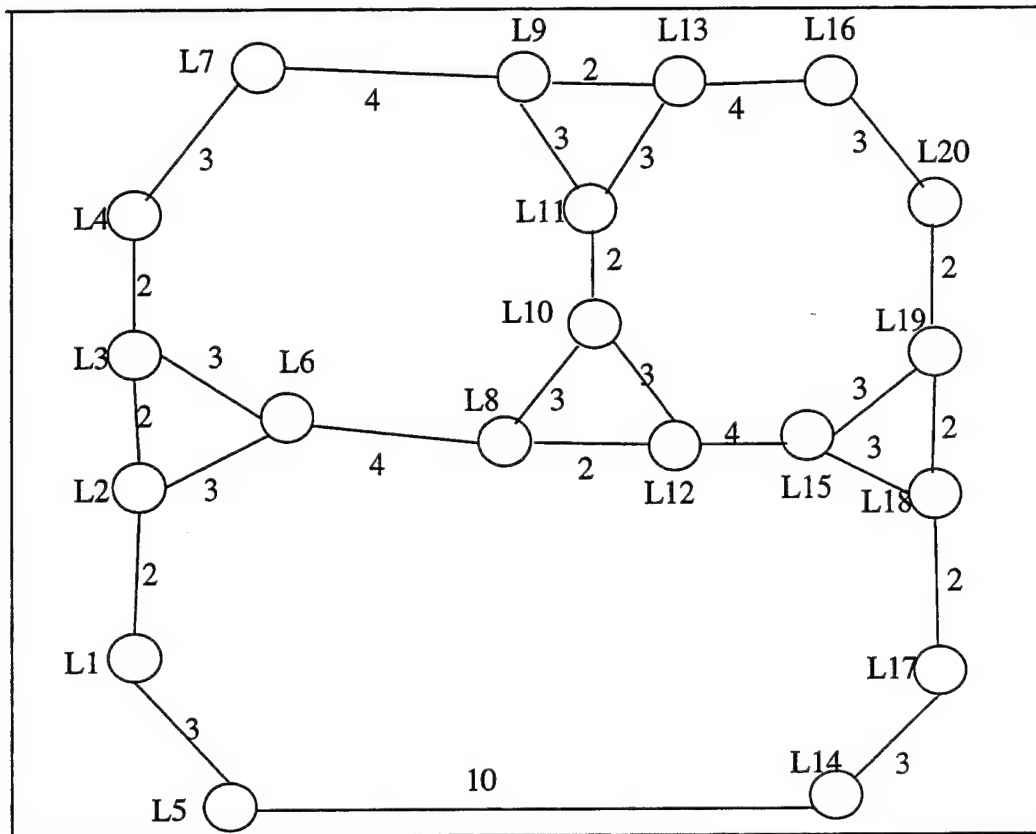


Figure 33. Basic Connectivity Graph

Given a start and a goal configurations, we add two new nodes, S and G, to the basic connectivity graph to obtain an *augmented connectivity graph*. A directed edge from S to each border in the start region is added to the graph. A cost of the edge is made equal to the energy needed for a robot to make a motion from the start configuration to the border. Likewise, a directed edge from each border in the goal region to G is added. A cost of the edge is made equal to the energy needed for a robot to make a motion from the border to the goal configuration. Figure 34. shows an augmented connected graph for the convexly decomposed world in Figure 32.

B. ALGORITHM DESCRIPTION

Now, finding the “optimal” path class from S to G in the world shown in Figure 32. is transformed into the minimum cost path finding problem from S to G in the augmented connectivity graph shown in Figure 34. The Dijkstra’s algorithm can be perfectly applied to this global motion planning problem. As a result, a border sequence is obtained. The computation time is $O(N^2)$ using two for loops with the one being nested, where N is the number of nodes (borders) in the augmented connected graph respectively.

```
Minimum_Cost_Path_Finding(G, s, g)
begin
  for all vertices v do
    v.mark:= 0;
    v.cost:= ∞
  s.cost:= 0;
  v := s;
  repeat
    v.mark:= 1;
    for all edges (v0, v) ∈ E do
      if v0.cost + c(v0, v) < v.cost then
        v.cost:= v0.cost + c(v0, v) ;
        v.prev:= v0 ;
        v0 := [ v such that v.cost is minimum among unmarked v ];
  until v0 := g;
```

Figure 35. Dijkstra’s Algorithm

In the Dijkstras's algorithm $G = (V, E)$ is the augmented connectivity graph with a node set V and a edge set E . A cost $c(u, w) > 0$ is given to each edge (u, w) . The problem to be solved is to find the minimum cost path from s to g and the cost itself.

For each node u , we use variables $u.mark$, $u.cost$, and $u.prev$ in this algorithm. $u.cost$ is the minimum cost from s to u known so far. $u.mark$ is set to 1 when $u.cost$ is known to be the minimum cost. $u.prev$ is a pointer to the node which is placed to the previous position in the shortest path from s to u . When an execution of this program terminates, the minimum cost is found in $g.cost$ and the shortest path is obtained in a reverse manner starting from $g.prev$.

C. DATA STRUCTURE

To completely determine a graph, we need to know the number of nodes in it, and the neighbors of each node. This observation will lead us to specify the proper data structure for our implementation of Dijkstra's algorithm.

The connectivity graph is represented by an array of nodes, $GNodes[numOfNodes]$. Each node in the graph is a "structure" and has as a member, an array of pointers to node, $adjArray[numOfNodes]$. This way we can keep track of the neighbors of each node. The details of the data structure can be found in Appendix A.

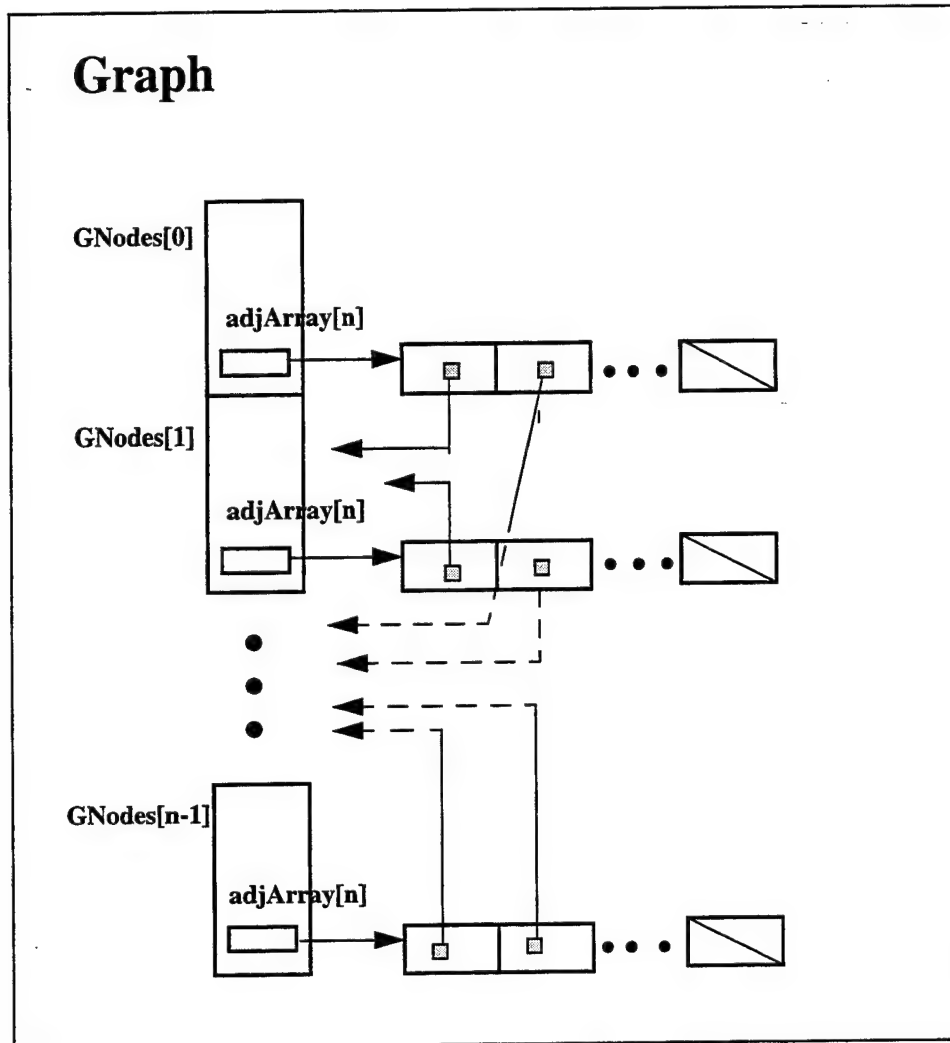


Figure 36. Dijkstra's Data Structure

D. IMPLEMENTATION

We applied Dijkstra's algorithm for the augmented connectivity graph in Figure 34. Figure 37. shows the nodes visited during the search. The minimum cost is found following the prev. pointers from the Goal node. There is only one path that leads to the Start node. Every node is visited (see bold circles). Inside each node the u.cost value is shown. Every

node has a not NULL previous pointer (u.prev). The nodes and the pointers for the minimum cost path are highlighted.

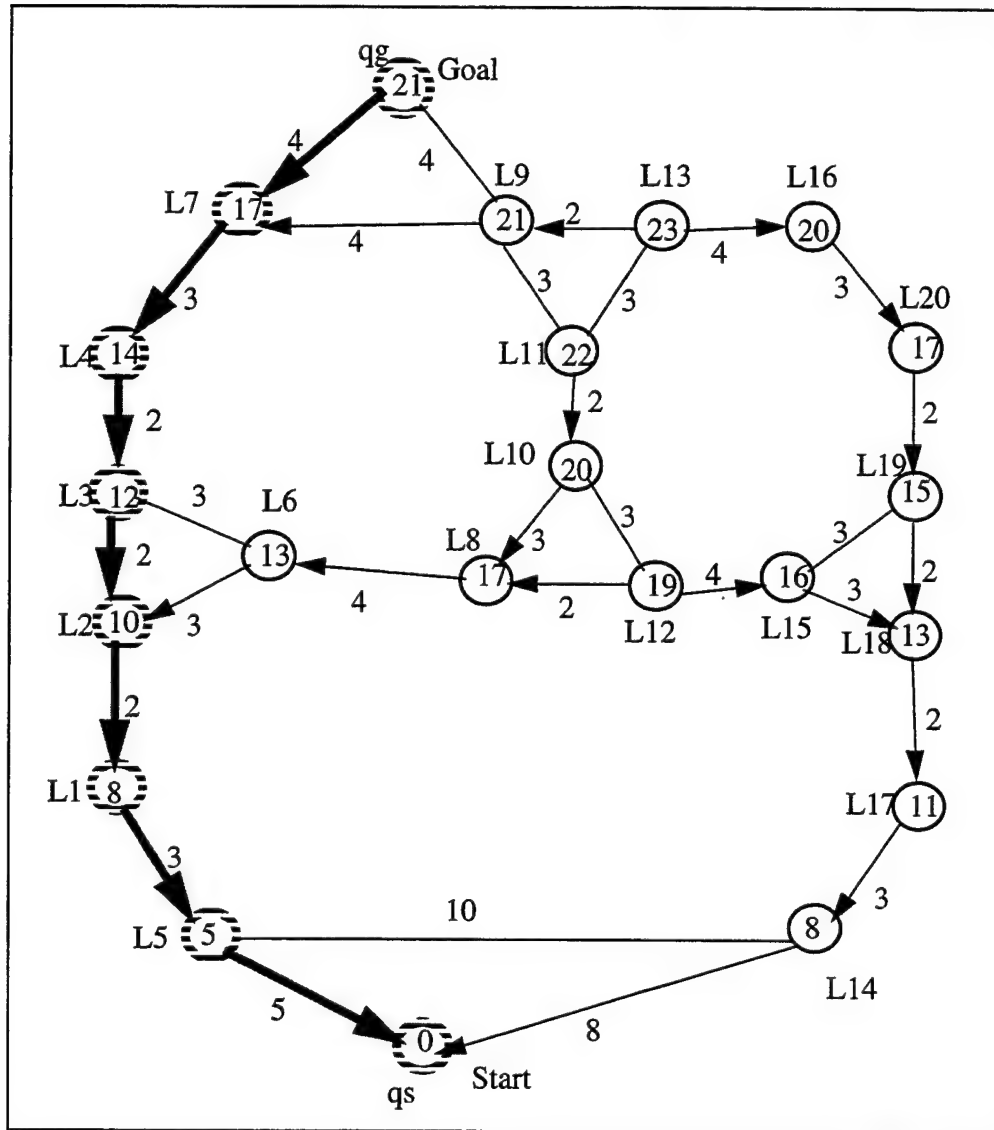


Figure 37. Dijkstra's Example Execution

IX. ALL - PAIRS MINIMUM COST PATHS ALGORITHM

A. CONNECTIVITY GRAPH OF THE WORLD

The problem to be solved is how to find the “optimal” path class? In order to solve this problem, we need to have a method to evaluate the cost of each path class. A reasonable approximation method looks translating the polygonal world into a graph [KAN95b].

Given a world W with a homotopic decomposition D , its *basic connectivity graph* $G = (V, E)$ is defined as follow: V is a set of nodes or vertices and E a set of edges.

$$V = \{L^+, L^- \mid (L \in D)\},$$

namely, we create nodes L^+ and L^- for each border in the decomposition.

$$E = \{(z_1, z_2) \mid \text{there exists an oriented region in } D \\ \text{such that its entrance is } z_1 \text{ and its exit is } z_2\}.$$

These edges are *directed*. A cost for this edge is computed, for instance, as the energy (or time) needed to move the vehicle from the center of the entrance border to the center of exit border with appropriate vehicle directions. In this cost evaluation, the cost for both translational and rotational motions should be included. The basic connectivity graph of a decomposition shown in the upper half of Figure 38., is translated into the basic connectivity graph shown in the lower half of the same Figure (a cost for each edge is not specifically shown).

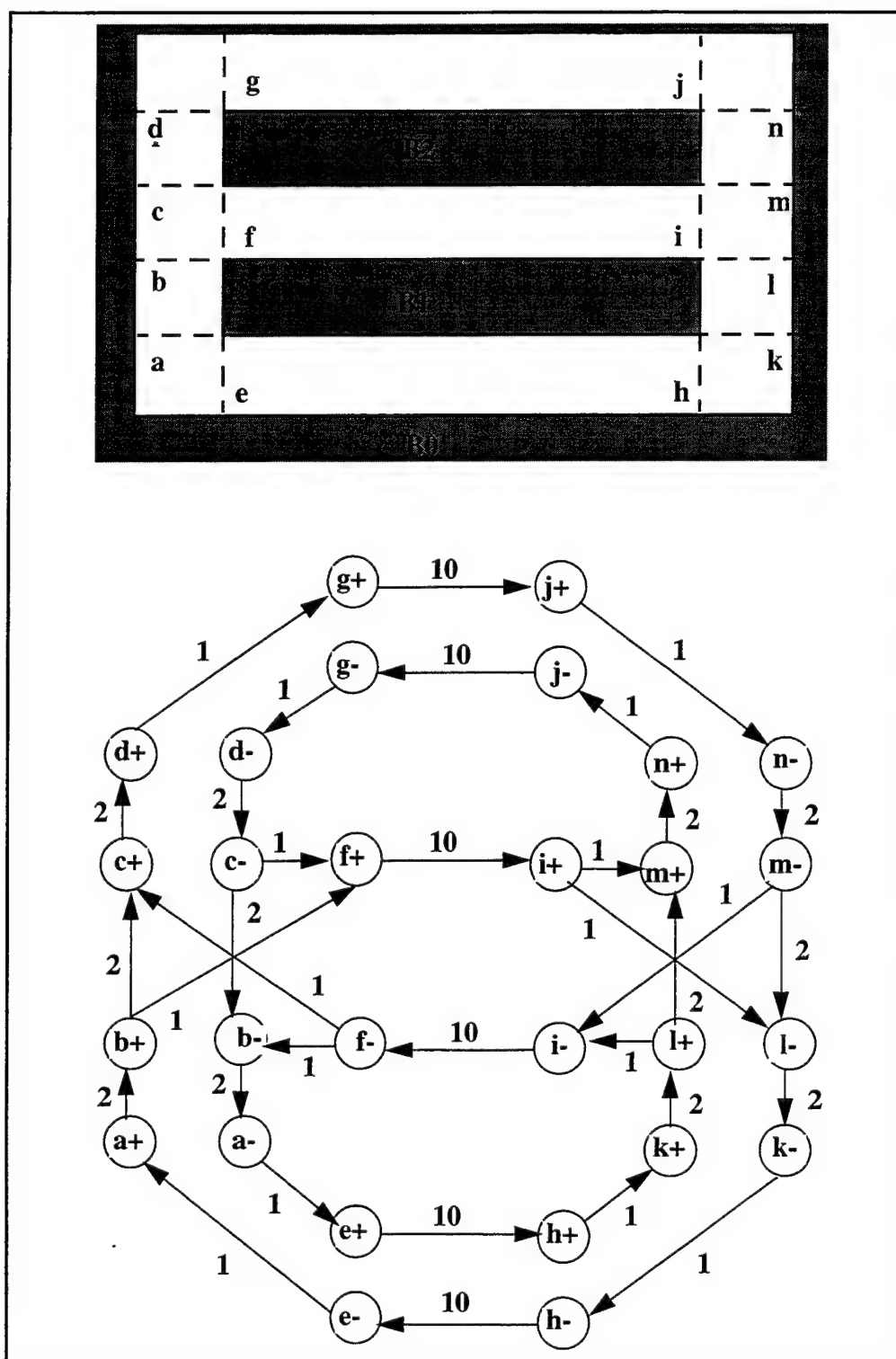


Figure 38. A Decomposition and Its Basic Connectivity Graph

It is straightforward to compute "all-pair minimum cost paths" on this graph G . By this method we preprocess the minimum cost from any oriented border to any other oriented border in $O(N^3)$, where N is the number of borders in D . After this preprocessing, for any oriented border pair (z, z') , the cost $c(z, z')$ is obtained in a constant time. In general, $c(z, z') \neq c(z', z)$.

So far this preprocessing is done independent of a situation with a given start-goal configuration (position and direction) pair. Given two end-configurations, q_s and q_g , we create an *augmented connectivity graph* based on the basic connectivity graph by the following procedure:

1. Add nodes q_s and q_g to V . An orientation + or - is not needed.
2. Let R_s be the region where q_s belongs to. Let z_{s1}, z_{s2}, \dots be the possible exit borders belonging to R_s . Add an edge (q_s, z_{si}) to the edge set E for each i . Evaluate the cost $c(q_s, z_{si})$ using some simulation technique and define it as the edge cost. This task is called *initial portion motion planning*.
3. Let R_g be the region where q_g belongs to. Let z_{g1}, z_{g2}, \dots be possible entrance borders belonging to R_g . Add an edge (z_{gi}, q_g) to the edge set E for each i . Evaluate the cost $c(z_{gi}, q_g)$ using some simulation technique and define it as the edge cost. This task is called *final portion motion planning*.

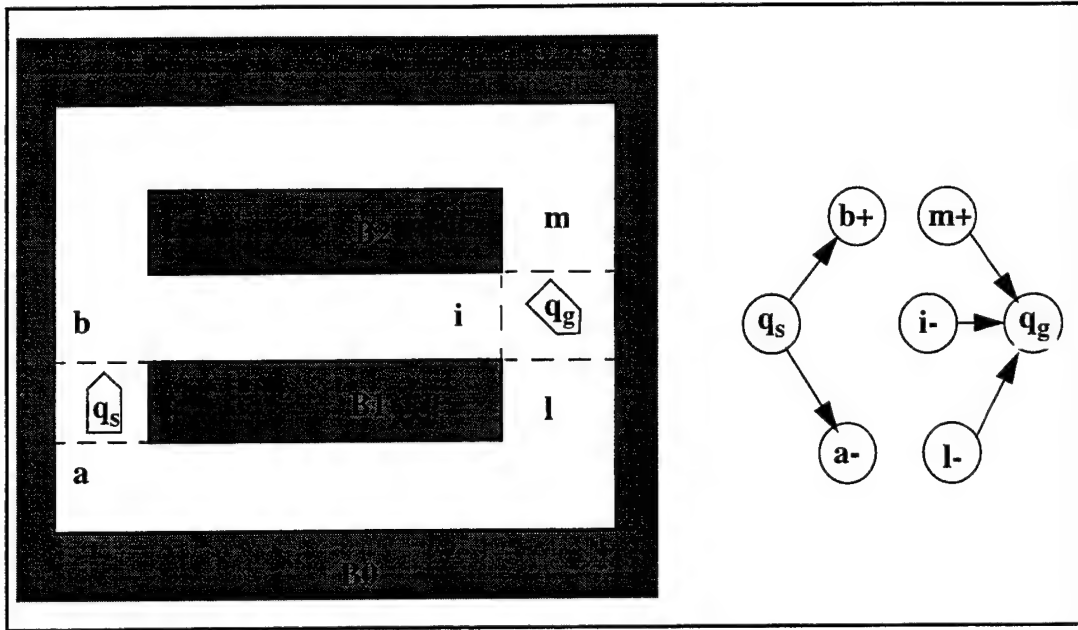


Figure 39. Addition for Augmented Connectivity Graph

If the start and goal configurations are specified as shown in Figure 39., we will add two nodes, q_s and q_g , and five edges - as shown also in the right half of the same Figure - to the basic connectivity graph. The result is an augmented connectivity graph.

Then the original global motion planning algorithm is transformed into a minimum cost path finding problem from q_s to q_g in the augmented graph. If the numbers of edges added in Steps 2 and 3 are M and K respectively, we need to compare MK distinct costs. In a given world, the maximum values of M and K are limited. Therefore the cost for the graph search portion is constant. The most time consuming task is the initial/final motion planning described in Steps 2 and 3. This is a good result when we recollect the fact that the number of distinct path classes is beyond any polynomial function of the number of obstacles n .

As a review, first we divide the free space of the world into regions using borders. A connectivity graph is next defined to represent the geometrical relation of the world and to represent distinct path classes symbolically. In this graph, the borders are nodes as

opposed the previous method, in which regions are used as nodes. A pair of borders (nodes), which belong to the same region, is connected by an edge in the connectivity graph. A cost of the motion between the two borders is evaluated by simulation and given to the corresponding edge. We use the notion of oriented borders for more descriptive representation of path classes in the connectivity graph. In order to find the optimal path class the All-Pairs Minimum Cost Paths algorithm is applied.

B. ALGORITHM DESCRIPTION

The problem: Given a *directed* weighted graph $G=(V, E)$ with nonnegative costs, find the minimum-cost paths between all pairs of vertices.

The solution: There is an induction method useful to solve the problem. We leave the number of vertices fixed, and we put restrictions on the type of paths allowed. The inductions addresses the removals of these restrictions on the paths until, at the end, all possible paths are considered. We label the vertices from 1 to $|V|$. A path from u to w is called a k -path if, except for u and w , the highest-label vertex on the path is labeled k . In particular, a 0-path is an edge (since no other vertices can appear on the path).

Induction hypothesis: We know the lengths of the minimum cost paths between all pairs of vertices such that only k -paths, for some $k < m$, are considered.

The base of the induction is $m=1$, in which case only direct edges can be considered and the solution is obvious. We assume the induction hypothesis for m , and we try to extend it to $m+1$.

Algorithm All-Pair Minimum Cost Paths

Input:

Cost (an $n * n$ adjacency matrix representing a weighted graph).

{*Cost* [*x*, *y*] is the cost of the edge (*x*, *y*) if it exists, and Infinity otherwise;

Cost [*x*, *x*] is 0, for all *x*}

Previous (an $n * n$ adjacency matrix containing vertices).

Initial state for *Previous*:

{*Previous* [*x*, *y*] is the id of vertex (*y*), if *Cost*[*x*,*y*]!= Infinity.

Previous [*x*, *y*] is Undefined otherwise, and Infinity is assigned to that entry. }

Output:

At the end matrix *Cost* contains the lengths of the shortest paths and matrix *Previous* contains the neighbor to (*x*) vertex, to be tracked in the route for the shortest paths, for each entry [*x*, *y*].

begin

 for *m* := 1 to *n* do

 for *x* := 1 to *n* do

 for *y* := 1 to *n* do

 if *Cost*[*x*, *m*] + *Cost*[*m*,*y*] < *Cost*[*x*, *y*] then

Cost[*x*, *y*] := *Cost*[*x*, *m*] + *Cost*[*m*,*y*]

Previous[*x*, *y*] := *Previous*[*x*, *m*]

end

Figure 40. All-Pairs Minimum Cost Paths algorithm

C. DATA STRUCTURE

To completely determine the graph, we need to define the nodes of it and the neighbors of each node. We are using a two-dimensional array, named $\text{Cost}[\text{numOfNodes}][\text{numOfNodes}]$. This array provides the information on the number of nodes, and the connectivity between them.

Let x, y nodes in the connectivity graph. Cost is 0 for an entry (x, y) of the Cost matrix, if $x = y$. Cost is ∞ for an entry (x, y) , if $x \neq y$ and the two nodes are not connected. For all other cases we assign a value representing the energy or time a robot needs to get from node x to node y of the graph.

Then we are using a two-dimensional array, named $\text{Previous}[\text{numOfNodes}][\text{numOfNodes}]$. This array provides the information on which is the actual neighbor of each node (by node-id), and will keep the information in the search for the minimum cost path between every single pair of nodes.

For an entry (x, y) of the Cost matrix, Previous node is assigned the y -node's id, if $x \neq y$ and the two nodes are connected. Previous is assigned 0 in all other cases.

The details of the data structure can be found in Appendix A.

D. IMPLEMENTATION

We applied All-Pairs Minimum Cost Paths algorithm for the connectivity graph in Figure 38. The input file contains the edge cost values for the nodes. When two nodes are not connected the value of 99 is used for ∞ .

The initial settings for the Cost and Previous arrays are shown in Figure 41. and Figure 42. respectively. ($D^{(0)}$ and $\Pi^{(0)}$ matrices).

The result Cost and Previous arrays are shown in Figure 43. and Figure 44. respectively. ($D^{(28)}$ and $\Pi^{(28)}$ matrices). The details of the implementation program can be found in Appendix A.

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 99 2 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
2 * 99 0 99 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
3 * 99 99 0 99 2 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
4 * 99 2 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
5 * 99 99 99 99 0 99 2 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
6 * 99 99 99 2 99 0 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
7 * 99 99 99 99 99 99 0 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99
8 * 99 99 99 99 99 2 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
9 * 99 99 99 99 99 99 99 99 0 99 99 99 99 99 10 99 99 99 99 99 99 99 99 99 99 99
10* 1 99 99 99 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
11* 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 10 99 99 99 99 99 99 99 99 99
12* 99 99 99 1 1 99 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99
13* 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 10 99 99 99 99 99 99 99 99
14* 99 99 99 99 99 99 99 99 1 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99
15* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 1 99 99 99 99 99 99 99
16* 99 99 99 99 99 99 99 99 99 99 10 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99
17* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 1 1 99 99 99
18* 99 99 99 99 99 99 99 99 99 99 99 99 10 99 99 99 99 0 99 99 99 99 99 99 99 99
19* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 99 99 1
20* 99 99 99 99 99 99 99 99 99 99 99 99 99 10 99 99 99 99 99 0 99 99 99 99 99 99
21* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 2 99 99 99 99 99 99
22* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 1 99 99 99 99 0 99 99 99 99
23* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 1 99 99 99 99 0 99 99
24* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 2 99 0 99 99 99 99
25* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 2 99
26* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 1 99 99 99 99 2 99 0 99 99
27* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99
28* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 2 99 0

```

Figure 41. Cost Array - Initial Matrix $D^{(0)}$

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 * 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 * 0 0 0 0 0 3 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 * 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 * 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 * 0 0 0 0 6 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0
7 * 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0
8 * 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0
10* 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0
12* 0 0 0 0 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 13 0 0 0 0 0 0 0
14* 0 0 0 0 0 0 0 0 0 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15 0 0 0 0 0 0
16* 0 0 0 0 0 0 0 0 0 0 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
17* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 17 17 0 0 0
18* 0 0 0 0 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 0
19* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 19
20* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 20 0 0 0 0 0 0 0 0 0 0 0
21* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0 0 0
22* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 22 0 0 0 0 0 0 0 0 0 0
23* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 23 0 0 0 0 0 0 0 0
24* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24 0 0 0 0 0 0 0
25* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 0 0 0 0 0 0
26* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 26 0 0 0 0 0
27* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 27 0 0 0 0 0
28* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 28 0 0 0

```

Figure 42. Previous Array - Initial Matrix $\Pi^{(0)}$

***	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	
1 *	0	34	2	32	4	30	6	28	35	27	3	31	7	27	45	17	13	21	17	17	46	16	48	14	14	20	16	18	
2 *	58	0	60	26	26	88	28	86	1	57	61	25	29	85	11	47	71	15	39	75	12	46	14	44	72	42	74	40	
3 *	26	32	0	30	2	28	4	26	33	25	1	29	5	25	43	15	11	19	15	15	44	14	46	12	12	18	14	16	
4 *	60	2	62	0	28	90	30	88	3	59	63	27	31	87	13	49	73	17	41	77	14	48	16	46	74	44	76	42	
5 *	32	30	34	28	0	62	2	60	31	31	35	27	3	59	41	21	45	17	13	49	42	20	44	18	46	16	48	14	
6 *	26	4	28	2	30	0	32	26	5	25	1	29	33	25	15	15	11	19	43	15	16	14	18	12	12	46	14	44	
7 *	30	28	32	26	26	60	0	58	29	29	33	25	1	57	39	19	43	15	11	47	40	18	42	16	44	14	46	12	
8 *	28	6	30	4	32	2	34	0	7	27	3	31	35	27	17	17	13	21	45	17	18	16	20	14	14	48	16	46	
9 *	57	27	59	25	25	87	27	85	0	56	60	24	28	84	10	46	70	14	38	74	11	45	13	43	71	41	73	39	
10*	1	35	3	33	5	31	7	29	36	0	4	32	8	28	46	18	14	22	18	18	47	17	49	15	15	21	17	19	
11*	25	31	27	29	29	27	31	25	32	24	0	56	32	24	42	14	10	46	42	14	43	13	45	11	11	45	13	43	
12*	33	3	35	1	1	63	3	61	4	32	36	0	4	60	14	22	46	18	14	50	15	21	17	19	47	17	49	15	
13*	29	27	31	25	25	59	27	57	28	32	24	0	56	38	18	42	14	10	46	39	17	41	15	43	13	45	11	11	
14*	29	7	31	5	33	3	35	1	8	28	4	32	36	0	18	18	14	22	46	18	19	17	21	15	15	49	17	47	
15*	47	17	49	15	15	77	17	75	18	46	50	14	18	74	0	36	60	4	28	64	1	35	3	33	61	31	63	29	
16*	11	45	13	43	15	41	17	39	46	10	14	42	18	38	56	0	24	32	28	28	57	27	59	25	25	31	27	29	
17*	15	21	17	19	19	17	21	15	22	14	14	18	46	22	14	32	4	0	36	32	4	33	3	35	1	1	35	3	33
18*	43	13	45	11	11	73	13	71	14	42	46	10	14	70	24	32	56	0	24	60	25	31	27	29	57	27	59	25	
19*	19	17	21	15	15	49	17	47	18	18	22	14	18	46	28	8	32	4	0	36	29	7	31	5	33	3	35	1	
20*	39	17	41	15	43	13	45	11	18	38	14	42	46	10	28	28	24	32	56	0	29	27	31	25	25	59	27	57	
21*	46	16	48	14	14	76	16	74	17	45	49	13	17	73	27	35	59	3	27	63	0	34	2	32	60	30	62	28	
22*	12	46	14	44	16	42	18	40	47	11	15	43	19	39	57	1	25	33	29	29	58	0	60	26	26	32	28	30	
23*	44	14	46	12	12	74	14	72	15	43	47	11	15	71	25	33	57	1	25	61	26	32	0	30	58	28	60	26	
24*	14	48	16	46	18	44	20	42	49	13	17	45	21	41	59	3	27	35	31	31	60	2	62	0	28	34	30	32	
25*	42	20	44	18	46	16	48	14	21	41	17	45	49	13	31	31	27	35	59	3	32	30	34	28	0	62	2	60	
26*	16	14	18	12	12	46	14	44	15	15	19	11	15	43	25	5	29	1	25	33	26	4	28	2	30	0	32	26	
27*	40	18	42	16	44	14	46	12	19	39	15	43	47	11	29	29	25	33	57	1	30	28	32	26	26	60	0	58	
28*	18	16	20	14	14	48	16	46	17	17	21	13	17	45	27	7	31	3	27	35	28	6	30	4	32	2	34	0	

Figure 43. Cost Array - Result Matrix $D^{(28)}$

***	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	
1	*	0	4	1	6	3	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	17	17	28	25	19
2	*	10	0	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	23	13	27	15	24	21	26	17	28	25	19
3	*	10	4	0	6	3	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	17	17	28	25	19
4	*	10	4	1	0	12	8	5	14	2	16	3	18	7	20	9	22	11	23	13	27	15	24	21	26	17	28	25	19
5	*	10	4	1	12	0	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	26	17	28	25	19
6	*	10	4	1	6	12	0	5	14	2	16	6	18	7	20	9	22	11	23	13	27	15	24	21	26	17	28	25	19
7	*	10	4	1	12	12	8	0	14	2	16	3	18	7	20	9	22	11	23	13	27	15	24	21	17	17	28	25	19
8	*	10	4	1	6	12	8	5	0	2	16	6	18	7	20	9	22	11	26	13	27	15	24	21	26	17	28	25	19
9	*	10	4	1	12	12	8	5	14	0	16	3	18	7	20	9	22	11	23	13	27	15	24	21	17	17	28	25	19
10	*	10	4	1	6	3	8	5	14	2	0	3	18	7	20	9	22	11	26	13	27	15	24	21	17	17	28	25	19
11	*	10	4	1	6	3	8	5	14	2	16	0	18	7	20	9	22	11	23	13	27	15	24	21	17	17	28	25	19
12	*	10	4	1	12	12	8	5	14	2	16	3	0	7	20	9	22	11	23	13	27	15	24	21	26	17	28	25	19
13	*	10	4	1	12	12	8	5	14	2	16	3	18	0	20	9	22	11	26	13	27	15	24	21	26	17	28	25	19
14	*	10	4	1	6	12	8	5	14	2	16	6	18	7	0	9	22	11	26	13	27	15	24	21	26	17	28	25	19
15	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	0	22	11	23	13	27	15	24	21	26	17	28	25	19
16	*	10	4	1	6	3	8	5	14	2	16	3	18	7	20	9	0	11	26	13	27	15	24	21	17	17	28	25	19
17	*	10	4	1	6	3	8	5	14	2	16	3	18	7	20	9	22	0	23	13	27	15	24	21	17	17	28	25	19
18	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	0	13	27	15	24	21	26	17	28	25	19
19	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	26	0	27	15	24	21	26	17	28	25	19
20	*	10	4	1	6	12	8	5	14	2	16	6	18	7	20	9	22	11	23	13	0	15	24	21	17	17	28	25	19
21	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	23	13	27	0	24	21	26	17	28	25	19
22	*	10	4	1	6	3	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	0	21	17	17	28	25	19
23	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	23	13	27	15	24	0	26	17	28	25	19
24	*	10	4	1	6	3	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	0	17	28	25	19
25	*	10	4	1	6	12	8	5	14	2	16	6	18	7	20	9	22	11	23	13	27	15	24	21	17	0	28	25	19
26	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	26	17	0	25	19
27	*	10	4	1	6	12	8	5	14	2	16	6	18	7	20	9	22	11	23	13	27	15	24	21	17	17	28	0	19
28	*	10	4	1	12	12	8	5	14	2	16	3	18	7	20	9	22	11	26	13	27	15	24	21	26	17	28	25	0

Figure 44. Previous Array - Result Matrix $\Pi^{(28)}$

X. CONCLUSION

A. RESULTS

1. Sonar Characteristics Measurement

Yamabico's basic sonar characteristics, taken by translational scanning, showed reasonable results for the sonars. The experiments taken by applying the left and right sonars. The results indicate the sonars have a high degree of accuracy while the robot is moving. Sonar data had a precision of 1 to 2.5 cm during the testing, in robot's distances from the object 100 cm to 150 cm. This is the difference between some dimension of an object in the real world and its image in the robot's understanding of this world.

The coordination is successful between the sonar and motion systems. Specifically, the linear fitting algorithm accurately applies sonar data to build line segments that represent the wall.

2. Global Motion Planning

The connectivity graph representing a model world was given as input to two existing algorithms; *Dijkstra's* and *All-Pairs Minimum Cost Paths*. For the physical environment used in *Yamabico* robot motion experiments, this connectivity graph is typically a sparse one.

To find a minimum cost path from a start to a goal configuration, *Dijkstra's* algorithm applied in this environment. Many unnecessary comparisons between each node's non-neighbor nodes were avoided. This algorithm has a time complexity of $O(n^2)$ and is to be used when the world is dynamic and n is very large; otherwise the *All-Pairs Minimum Cost Paths* algorithm with time complexity $O(n^3)$, is far better because preprocessing makes the path class finding time to $O(1)$.

A known static world favors the use of pre-calculated minimum cost paths between all pairs of nodes in the respective basic connectivity graph. Thus, the *All-Pairs Minimum Cost* algorithm is ideally suited for the *Yamabico* operating environment. For this

environment, the time complexity $O(n^3)$ using the *All-Pairs Minimum Cost* algorithm is better to that applying the *Dijkstra's* algorithm.

The latter still has a chance when the graph is dense. We will prefer to use it, to avoid memory space problems that are likely to occur in the first algorithm's implementation. Thus, *All-Pairs Minimum Cost* algorithm is the optimal method for global motion planning.

APPENDIX A. USER PROGRAMMS

This appendix contains the program files which describe the algorithms implemented by this work. Inputs and outputs are described accordingly. The heading of each file explains the details.

```

/*****
*
* FILE      : dijkstra.C
* AUTHOR    : Athanassios Papadatos
* DATE      : 31 October 1995
* DESCRIPTION: Minimum Cost Path Finding. Applies to Robot
*            Global Motion Planning.
*****/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

#define INFINITY 99

#define numOfNodes 22 //HERE # OF NODES

#define out_file "dijkstra.dat"

ofstream kds;

//adjArray is an array of pointers to nodes.
//Each pointer ( adjArray[i] ) points to the address
//of node GNodes[i], to be initialized when
//build the Graph.

struct Node {
    int id;
    int cost;
    int mark; // its value is "0" or "1"
    Node *prev; // a pointer to previous node
    Node *adjArray[numOfNodes];
};

//a Graph is an array of nodes & an array of edges represented by their cost.
//Start - Goal node are included in CGraph.
//an Edge is represented by [i][j] the GNodes, and by its cost.

struct Graph {
    Node GNodes[numOfNodes];
    int ECost[numOfNodes][numOfNodes];
};

```

```

/*****
* FUNCTION: extractMin(Graph G)
* DESCRIPTION: Searches for the node in GNodes[] with min cost
*               an Returns its index.
*****/

int
extractMin(Graph G)
{
    Node tmp;

    tmp = G.GNodes[numOfNodes-1]; //if nothing the last one is the min!!!!

    int min= numOfNodes-1; //at begining last element is min OR...
    for (int j=0; j<numOfNodes; j++) {
        if (G.GNodes[j].mark == 0 ) {
            if ( G.GNodes[j].cost < tmp.cost ) {
                tmp = G.GNodes[j];
                min=j;
            }
        }
    }

    return (min);
}

```

```

/*****
*
* FUNCTION      :Dijkstra()
* DESCRIPTION : Applies the main Dijkstra's algorithm
*
*****/

void
Dijkstra()
{
    Graph G;

    ifstream in_file("graph22.dat", ios::in);

```

```

    if(!in_file){
        cerr << "File could not be opened" << endl;
        exit(1);
    }

    /*******
    * COMMENTS: In this part for each node record we initialize all its fields.
    *            that is: cost, mark, *prev.
    *            *****/

    //Initialize all nodes.
    for (int n=0; n<numOfNodes; n++) {

        G.GNodes[n].id = n; //if the first node then cost is 0
        G.GNodes[n].cost = INFINITY; //n ? INFINITY : 0;
        G.GNodes[n].mark = 0;
        G.GNodes[n].prev = 0;

    }

    kds<<endl;

    /*******
    * COMMENTS : In this part of the code we initialize Edge Costs.
    *            It initializes a 2-diamensional array to G.EDCost[i][j]= cost
    *            thus for the edge comprised by the ith-jth Nodes we'll have its cost.
    *            *****/

    //When looking for 0's neighbors it has one if there is edge 01.
    //When looking for 1's neighbors it has one if there is edge 10.

    for (int i=0; i<numOfNodes; i++) {
        for (int j=0; j<numOfNodes; j++) {

            int cost;

            in_file >> cost;
            G.EDCost[i][j]= cost;

        }
    }

```



```

/*****
*COMMENTS : We Initialize ADJ. ARRAYS
*               Per each node, one array of pointers to its neighbors
*****/

```

```

    for (int b=0; b<numOfNodes; b++) {
        int w=0;
        for (int c=0; c<numOfNodes; c++) {
            if ( G.ECost[b][c] != INFINITY ) {

                G.GNodes[b].adjArray[w] = & G.GNodes[c];

                w++;

            }
            for (int d=w+1; d<numOfNodes; d++) {
                G.GNodes[b].adjArray[d] = NULL;
            }
        }
    }
}

```

```

//Here I assume I've already build the Graph with Start node = #0
//and goal = #(numOfNodes-1)
//All the nodes have cost= 9.
// mark, *prev, adjArrays initialized by run time when
// I'll define the edges.
//Let's begin the DIJKSTRA.

```

```

    //this is the start node
    G.GNodes[0].cost=0;

```

```

    int s=0;
    //find neighbors
    do {

```

```

        kds << "\n Min cost node : "<< s << ", MARK IT!!" << endl;
        kds << "#####" << endl;

```

```

        G.GNodes[s].mark= 1;

```

```

        //pick its adjArray of neighbors and for each neighbor...

```

```

        int j=0;
        while (G.GNodes[s].adjArray[j]) { //I reduce the time

```

```

int ind;
ind = (*G.GNodes[s].adjArray[j]).id; //the id of neighbor

int edgeCost = G.EdgeCost[s][ind];

kds << "\n" << s << "s neighbor is: " << ind << endl;
kds << "-----" << endl;

if (G.GNodes[s].cost + edgeCost < (*G.GNodes[s].adjArray[j]).cost) {
    (*G.GNodes[s].adjArray[j]).cost = G.GNodes[s].cost + edgeCost;
    (*G.GNodes[s].adjArray[j]).prev = &G.GNodes[s];

    kds << "Dijkstra's step DONE..." << endl;
    kds << ind << "s cost = " << G.GNodes[ind].cost << endl;
    kds << ind << "s prev ----> " << (*G.GNodes[ind].prev).id
<< endl;

}
else {

    kds << "Dijkstra's step DONE..." << endl;
    kds << ind << "has a smaller cost already!!!" << endl;

}

j++;

} //finish with this "s" node's neighbors

kds << endl << endl;

//Pick Minimum Cost Node
s = extractMin(G); //new neighbor

} while ( G.GNodes[s].id != numOfNodes-1 ); //GOAL NODE IS numOfNodes-1

//untill goal is reached ASSUME GOAL
//is the last node G.GNodes[numOfNodes-1].id

G.GNodes[numOfNodes-1].mark = 1; //MARK GOAL node

kds << "\n\nMin cost node is GOAL. Path determined!!!" << endl;
kds << "#####" << endl;
kds << "\nCost of Shortest Path = " << G.GNodes[numOfNodes-1].cost << endl;

```

```

    kds << "\n\n Shortest Path is: ";

    Node * tmp;
    tmp = & G.GNodes[numOfNodes-1]; //Initially, tmp points to the GOAL node

    kds << (*tmp).id;

    while ( (*tmp).prev != 0 ) {
        kds << " ---> ";

        kds << (*(tmp).prev).id;

        tmp = (*tmp).prev;
    }

    kds << endl << endl;

} //END Dijkstra(G)


/*****
*
* Function: main()
* Description: Makes the necessary calls for Dijkstra's algorithm
*              to execute.
*****/

int
main()
{
    kds.open(out_file);

    Dijkstra();

    kds.close();

    return(0);
}

```

```

/*****
*
* FILE      : ap.C
* AUTHOR    : Athanassios Papadatos
* DATE      : 1 Dec. 1995
* DESCRIPTION : All-Pairs Minimum Cost Paths Algorithm
*                                           : Applies to Robot Global
Motion Planning.
*
*****/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

#define INFINITY 99

#define numOfNodes 28    //HERE # OF NODES

#define out_file "ap.dat"

ofstream kds;

struct Node {
    int id;
};

//a Graph is an array of nodes & an array of edges
//represented by their cost.
//Start - Goal node are included in CGraph.
struct Graph {
    Node Nodes[numOfNodes];

    int Cost[numOfNodes][numOfNodes];
        //an Edge is represented by [i][j] the GNodes,
        //and by its cost.
    int Previous[numOfNodes][numOfNodes];
};

//Input x, y = 1..numOfNodes
void
traceSPath(Graph G, int x, int y)
{

```

```

kds<<"\n\n Shortest Path "<<y<<"<--"<<x<<"."<<endl;
kds<<"-----"<<endl;
kds<<y;

x=x-1;
y=y-1;
if ( x!=y) {
    while (1) {
        int tmp;

        //Previous returns node.id=1,...,n.
        tmp=G.Previous[x][y];

        kds <<"<--"<<tmp;

        y=(tmp-1);

        if ( x==y )
            break;
    }
}

}

void
allPairs()
{
    Graph G;

    ifstream in_file("graph28.dat", ios::in);
    if (!in_file) {
        cerr << "File could not be opened" << endl;
        exit(1);
    }

    /*****
    //      INITIALIZE NODES
    *****/

    for (int n=0; n<numOfNodes; n++) {
        G.Nodes[n].id = n+1;
    }
}

```

```

/*****
//      INITIALIZE Cost / Previous ARRAYS.

for (int i=0; i<numOfNodes; i++) {
    for (int j=0; j<numOfNodes; j++) {

        int weight;
        in_file >> weight;

        G.Cost[i][j]= weight;

        if ( (weight == INFINITY) || ( i == j ) ){

            G.Previous[i][j] = 0;
            //The above 0 stands for NIL predecessor.
        }
        else

            G.Previous[i][j] = G.Nodes[i].id;
    }
}

{

    kds<< "**** "<<setiosflags(ios::left)<<setw(3)<<"1"<<setw(3)<<"2"

<<setw(3)<<"3"<<setw(3)<<"4"<<setw(3)<<"5"<<setw(3)<<"6"<<setw(3)<<
"7"<<setw(3)<<"8"<<setw(3)<<"9"<<setw(3)<<"10"<<setw(3)<<"11"<<setw
(3)<<"12"<<setw(3)<<"13"<<setw(3)<<"14"<<setw(3)<<"15"<<setw(3)<<"1
6"

<<setw(3)<<"17"<<setw(3)<<"18"<<setw(3)<<"19"<<setw(3)<<"20"<<setw(3)
)<<"21"<<setw(3)<<"22"<<setw(3)<<"23"<<setw(3)<<"24"<<setw(3)<<"25"
<<setw(3)<<"26"<<setw(3)<<"27"<<setw(3)<<"28"<<endl;

kds<< "*****",
*****
    for (int a=0; a<numOfNodes; a++){
        kds<<endl;
        kds<<setiosflags(ios::left)<<setw(2)<<a+1<<"* ";
        for (int y=0; y<numOfNodes; y++){
            kds<<setiosflags(ios::left)<<setw(3)<<G.Cost[a][y];

        }
    }
}

```

```

kds<<endl<<endl;

kds<<"*** "<<setiosflags(ios::left)<<setw(3)<<"1"<<setw(3)<<"2"
<<setw(3)<<"3"<<setw(3)<<"4"<<setw(3)<<"5"<<setw(3)<<"6"<<setw(3)<<
"7"<<setw(3)<<"8"<<setw(3)<<"9"<<setw(3)<<"10"<<setw(3)<<"11"<<setw
(3)<<"12"<<setw(3)<<"13"<<setw(3)<<"14"<<setw(3)<<"15"<<setw(3)<<"1
6"

<<setw(3)<<"17"<<setw(3)<<"18"<<setw(3)<<"19"<<setw(3)<<"20"<<setw(3
)<<"21"<<setw(3)<<"22"<<setw(3)<<"23"<<setw(3)<<"24"<<setw(3)<<"25"
<<setw(3)<<"26"<<setw(3)<<"27"<<setw(3)<<"28"<<endl;

kds<<"*****
*****";

    for (int c=0; c<numOfNodes; c++){
        kds<<endl;
        kds<<setiosflags(ios::left)<<setw(2)<<c+1<<"* ";
        for (int d=0; d<numOfNodes; d++){
            kds<<setiosflags(ios::left)<<setw(3)<<G.Previous[c][d];

        }
    }

    kds<<endl<<endl;
    /*****

//Change Costs - lengths of shortest paths - and
// Previous

    for (int m=0; m<numOfNodes; m++) {
        for (int x=0; x<numOfNodes; x++) {
            for (int y=0; y<numOfNodes; y++) {

                if ( G.Cost[x][m] + G.Cost[m][y] < G.Cost[x][y] ) {
                    if ( G.Cost[x][m]!=INFINITY && G.Cost[m][y]!=INFINITY ) {

                        G.Cost[x][y] = G.Cost[x][m] + G.Cost[m][y];

                        G.Previous[x][y] = G.Previous[m][y];

                    }
                }
            }
        }
    }
    else {

```

```

        G.Cost[x][y] = G.Cost[x][y];
        G.Previous[x][y] = G.Previous[x][y];
    }
}

kds<<endl<<endl<<endl<<endl;

}

```

```

kds<<"*** "<<setiosflags(ios::left)<<setw(3)<<"1"<<setw(3)<<"2"

<<setw(3)<<"3"<<setw(3)<<"4"<<setw(3)<<"5"<<setw(3)<<"6"<<setw(3)<<
"7"<<setw(3)<<"8"<<setw(3)<<"9"<<setw(3)<<"10"<<setw(3)<<"11"<<setw
(3)<<"12"<<setw(3)<<"13"<<setw(3)<<"14"<<setw(3)<<"15"<<setw(3)<<"1
6"

<<setw(3)<<"17"<<setw(3)<<"18"<<setw(3)<<"19"<<setw(3)<<"20"<<setw(3)
)<<"21"<<setw(3)<<"22"<<setw(3)<<"23"<<setw(3)<<"24"<<setw(3)<<"25"
<<setw(3)<<"26"<<setw(3)<<"27"<<setw(3)<<"28"<<endl;

kds<<"*****
*****";

    for (int a=0; a<numOfNodes; a++){
        kds<<endl;
        kds<<setiosflags(ios::left)<<setw(2)<<a+1<<"* ";
        for (int y=0; y<numOfNodes; y++){
            kds<<setiosflags(ios::left)<<setw(3)<<G.Cost[a][y];

        }
    }
}

```

```

kds<<endl<<endl;

```

```

kds<<"*** "<<setiosflags(ios::left)<<setw(3)<<"1"<<setw(3)<<"2"

<<setw(3)<<"3"<<setw(3)<<"4"<<setw(3)<<"5"<<setw(3)<<"6"<<setw(3)<<

```



```

"7"<<setw(3)<<"8"<<setw(3)<<"9"<<setw(3)<<"10"<<setw(3)<<"11"<<setw(3)<<"12"<<setw(3)<<"13"<<setw(3)<<"14"<<setw(3)<<"15"<<setw(3)<<"16"

```

```

<<setw(3)<<"17"<<setw(3)<<"18"<<setw(3)<<"19"<<setw(3)<<"20"<<setw(3)<<"21"<<setw(3)<<"22"<<setw(3)<<"23"<<setw(3)<<"24"<<setw(3)<<"25"
<<setw(3)<<"26"<<setw(3)<<"27"<<setw(3)<<"28"<<endl;

```

```

kds<<"*****";

```

```

    for (int c=0; c<numOfNodes; c++){
        kds<<endl;
        kds<<setiosflags(ios::left)<<setw(2)<<c+1<<" ";
        for (int d=0; d<numOfNodes; d++){
            kds<<setiosflags(ios::left)<<setw(3)<<G.Previous[c][d];

```

```

        }
    }

```

```

    traceSPath(G, 1, 7);
    traceSPath(G, 1, 11);
    traceSPath(G, 1, 10);
    traceSPath(G, 1, 27);

```

```

    kds<<endl<<endl;
}

```

```

/*****
*FUNCTION      : main()
*DESCRIPTION   : Makes the call to function allPairs()
*               and arranges to open and close the
*               appropriate files.
*****/

```

```

int
main()
{

```

```

    kds.open(out_file);

```

```

        allPairs();

```

```

    kds.close();

```

```

    return(0);

```

```

}

```


APPENDIX B. PROGRAMMS' RESULTS

This appendix contains the output data files resulting from the implementation of algorithms described in Appendix A. The heading of each file explains the details.

```

/*****
*
* FILE      : dijkstra.dat
* AUTHOR    : Athanassios Papadatos
* DATE      : 31 October 1995
* DESCRIPTION: Contains the results of the search for the minimum cost
*              path for the graph in Chapter 8.
*****/

```

```

Min cost node : 0, MARK IT!!
#####

```

```

0's neighbor is: 5
-----
Dijkstra's step DONE...
5's cost = 5
5's prev ----> 0

```

```

0's neighbor is: 14
-----
Dijkstra's step DONE...
14's cost = 8
14's prev ----> 0

```

```

Min cost node : 5, MARK IT!!
#####

```

```

5's neighbor is: 0
-----
Dijkstra's step DONE...
0 has a smaller cost already!!!

```

```

5's neighbor is: 1
-----
Dijkstra's step DONE...
1's cost = 8
1's prev ----> 5

```

```

5's neighbor is: 14
-----
Dijkstra's step DONE...
14 has a smaller cost already!!!

```

```

Min cost node : 1, MARK IT!!
#####

```

1's neighbor is: 2

 Dijkstra's step DONE...
 2's cost = 10
 2's prev ----> 1

 1's neighbor is: 5

 Dijkstra's step DONE...
 5 has a smaller cost already!!!

Min cost node : 14, MARK IT!!
 #####

14's neighbor is: 0

 Dijkstra's step DONE...
 0 has a smaller cost already!!!

 14's neighbor is: 5

 Dijkstra's step DONE...
 5 has a smaller cost already!!!

 14's neighbor is: 17

 Dijkstra's step DONE...
 17's cost = 11
 17's prev ----> 14

Min cost node : 2, MARK IT!!
 #####

2's neighbor is: 1

 Dijkstra's step DONE...
 1 has a smaller cost already!!!

 2's neighbor is: 3

 Dijkstra's step DONE...
 3's cost = 12
 3's prev ----> 2

 2's neighbor is: 6

 Dijkstra's step DONE...
 6's cost = 13
 6's prev ----> 2

Min cost node : 17, MARK IT!!
 #####

17's neighbor is: 14

Dijkstra's step DONE...
14 has a smaller cost already!!!

17's neighbor is: 18

Dijkstra's step DONE...
18's cost = 13
18's prev ----> 17

Min cost node : 3, MARK IT!!
#####

3's neighbor is: 2

Dijkstra's step DONE...
2 has a smaller cost already!!!

3's neighbor is: 4

Dijkstra's step DONE...
4's cost = 14
4's prev ----> 3

3's neighbor is: 6

Dijkstra's step DONE...
6 has a smaller cost already!!!

Min cost node : 6, MARK IT!!
#####

6's neighbor is: 2

Dijkstra's step DONE...
2 has a smaller cost already!!!

6's neighbor is: 3

Dijkstra's step DONE...
3 has a smaller cost already!!!

6's neighbor is: 8

Dijkstra's step DONE...
8's cost = 17
8's prev ----> 6

Min cost node : 18, MARK IT!!
#####

18's neighbor is: 15

Dijkstra's step DONE...
15's cost = 16
15's prev ----> 18

18's neighbor is: 17

Dijkstra's step DONE...
17 has a smaller cost already!!!

18's neighbor is: 19

Dijkstra's step DONE...
19's cost = 15
19's prev ----> 18

Min cost node : 4, MARK IT!!

#####

4's neighbor is: 3

Dijkstra's step DONE...
3 has a smaller cost already!!!

4's neighbor is: 7

Dijkstra's step DONE...
7's cost = 17
7's prev ----> 4

Min cost node : 19, MARK IT!!

#####

19's neighbor is: 15

Dijkstra's step DONE...
15 has a smaller cost already!!!

19's neighbor is: 18

Dijkstra's step DONE...
18 has a smaller cost already!!!

19's neighbor is: 20

Dijkstra's step DONE...
20's cost = 17
20's prev ----> 19

Min cost node : 15, MARK IT!!

#####

15's neighbor is: 12

Dijkstra's step DONE...
12's cost = 20
12's prev ----> 15

15's neighbor is: 18

Dijkstra's step DONE...
18 has a smaller cost already!!!

15's neighbor is: 19

Dijkstra's step DONE...
19 has a smaller cost already!!!

Min cost node : 7, MARK IT!!
#####

7's neighbor is: 4

Dijkstra's step DONE...
4 has a smaller cost already!!!

7's neighbor is: 9

Dijkstra's step DONE...
9's cost = 21
9's prev ----> 7

7's neighbor is: 21

Dijkstra's step DONE...
21's cost = 21
21's prev ----> 7

Min cost node : 8, MARK IT!!
#####

8's neighbor is: 6

Dijkstra's step DONE...
6 has a smaller cost already!!!

8's neighbor is: 10

Dijkstra's step DONE...
10's cost = 20
10's prev ----> 8

8's neighbor is: 12

Dijkstra's step DONE...
12's cost = 19
12's prev ----> 8

Min cost node : 20, MARK IT!!
#####

20's neighbor is: 16

Dijkstra's step DONE...

16's cost = 20
16's prev ----> 20

20's neighbor is: 19

Dijkstra's step DONE...
19 has a smaller cost already!!!

Min cost node : 12, MARK IT!!
#####

12's neighbor is: 8

Dijkstra's step DONE...
8 has a smaller cost already!!!

12's neighbor is: 10

Dijkstra's step DONE...
10 has a smaller cost already!!!

12's neighbor is: 15

Dijkstra's step DONE...
15 has a smaller cost already!!!

Min cost node : 10, MARK IT!!
#####

10's neighbor is: 8

Dijkstra's step DONE...
8 has a smaller cost already!!!

10's neighbor is: 11

Dijkstra's step DONE...
11's cost = 22
11's prev ----> 10

10's neighbor is: 12

Dijkstra's step DONE...
12 has a smaller cost already!!!

Min cost node : 16, MARK IT!!
#####

16's neighbor is: 13

Dijkstra's step DONE...
13's cost = 24
13's prev ----> 16

16's neighbor is: 20

Dijkstra's step DONE...
20 has a smaller cost already!!!

Min cost node is GOAL. Path determined!!!
#####

Cost of Shortest Path =21

Shortest Path is: 21 ---> 7 ---> 4 ---> 3 ---> 2 ---> 1 ---> 5 ---> 0

```

/*****
*
* FILE      : ap.dat
* AUTHOR    : Athanassios Papadatos
* DATE      : 31 October 1995
* DESCRIPTION: Contains the result output for All-Pairs Minimum Cost Paths
*            algorithm for the graph in chapter 9.
*****/

```

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 99 2 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
2 * 99 0 99 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
3 * 99 99 0 99 2 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
4 * 99 2 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
5 * 99 99 99 99 0 99 2 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
6 * 99 99 99 2 99 0 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
7 * 99 99 99 99 99 99 0 99 99 99 99 99 1 99 99 99 99 99 99 99 99 99 99 99 99 99
8 * 99 99 99 99 99 2 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
9 * 99 99 99 99 99 99 99 99 0 99 99 99 99 99 10 99 99 99 99 99 99 99 99 99 99 99
10* 1 99 99 99 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
11* 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 10 99 99 99 99 99 99 99 99 99
12* 99 99 99 1 1 99 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99
13* 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 10 99 99 99 99 99 99 99
14* 99 99 99 99 99 99 99 99 1 99 99 99 99 99 0 99 99 99 99 99 99 99 99 99 99 99 99
15* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 1 99 99 99 99 99
16* 99 99 99 99 99 99 99 99 99 99 99 99 10 99 99 99 99 99 0 99 99 99 99 99 99 99 99
17* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 0 99 99 99 99 99 1 1 99 99
18* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
19* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 1
20* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
21* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
22* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 1 99 99 99 99 99 0 99 99 99 99
23* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
24* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
25* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
26* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
27* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
28* 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99

```

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 * 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 * 0 0 0 0 3 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 * 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 * 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 * 0 0 0 6 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 * 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0
8 * 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0
10* 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0 0
12* 0 0 0 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 13 0 0 0 0 0 0 0 0
14* 0 0 0 0 0 0 0 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15 0 0 0 0 0 0 0
16* 0 0 0 0 0 0 0 0 0 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
17* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 17 17 0 0
18* 0 0 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 0 0
19* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 19
20* 0 0 0 0 0 0 0 0 0 0 0 0 20 0 0 0 0 0 0 0 0 0 0 0 0
21* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0 0 0 0
22* 0 0 0 0 0 0 0 0 0 0 0 0 0 22 0 0 0 0 0 0 0 0 0 0 0
23* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 23 0 0 0 0 0 0 0 0 0
24* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24 0 0 0 0 0 0 0
25* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 0
26* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 26 0 0 0 0 26 0 0 0
27* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 27 0 0 0 0 0 0 0
28* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 28 0 0

```

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 34 2 32 4 30 6 28 35 27 3 31 7 27 45 17 13 21 17 17 46 16 48 14 14 20 16 18
2 * 58 0 60 26 26 88 28 86 1 57 61 25 29 85 11 47 71 15 39 75 12 46 14 44 72 42 74 40
3 * 26 32 0 30 2 28 4 26 33 25 1 29 5 25 43 15 11 19 15 15 44 14 46 12 12 18 14 16
4 * 60 2 62 0 28 90 30 88 3 59 63 27 31 87 13 49 73 17 41 77 14 48 16 46 74 44 76 42
5 * 32 30 34 28 0 62 2 60 31 31 35 27 3 59 41 21 45 17 13 49 42 20 44 18 46 16 48 14
6 * 26 4 28 2 30 0 32 26 5 25 1 29 33 25 15 15 11 19 43 15 16 14 18 12 12 46 14 44
7 * 30 28 32 26 26 60 0 58 29 29 33 25 1 57 39 19 43 15 11 47 40 18 42 16 44 14 46 12
8 * 28 6 30 4 32 2 34 0 7 27 3 31 35 27 17 17 13 21 45 17 18 16 20 14 14 48 16 46
9 * 57 27 59 25 25 87 27 85 0 56 60 24 28 84 10 46 70 14 38 74 11 45 13 43 71 41 73 39
10* 1 35 3 33 5 31 7 29 36 0 4 32 8 28 46 18 14 22 18 18 47 17 49 15 15 21 17 19
11* 25 31 27 29 29 27 31 25 32 24 0 56 32 24 42 14 10 46 42 14 43 13 45 11 11 45 13 43
12* 33 3 35 1 1 63 3 61 4 32 36 0 4 60 14 22 46 18 14 50 15 21 17 19 47 17 49 15
13* 29 27 31 25 25 59 27 57 28 28 32 24 0 56 38 18 42 14 10 46 39 17 41 15 43 13 45 11
14* 29 7 31 5 33 3 35 1 8 28 4 32 36 0 18 18 14 22 46 18 19 17 21 15 15 49 17 47
15* 47 17 49 15 15 77 17 75 18 46 50 14 18 74 0 36 60 4 28 64 1 35 3 33 61 31 63 29
16* 11 45 13 43 15 41 17 39 46 10 14 42 18 38 56 0 24 32 28 28 57 27 59 25 25 31 27 29
17* 15 21 17 19 19 17 21 15 22 14 18 46 22 14 32 4 0 36 32 4 33 3 35 1 1 35 3 33
18* 43 13 45 11 11 73 13 71 14 42 46 10 14 70 24 32 56 0 24 60 25 31 27 29 57 27 59 25
19* 19 17 21 15 15 49 17 47 18 18 22 14 18 46 28 8 32 4 0 36 29 7 31 5 33 3 35 1
20* 39 17 41 15 43 13 45 11 18 38 14 42 46 10 28 28 24 32 56 0 29 27 31 25 25 59 27 57
21* 46 16 48 14 14 76 16 74 17 45 49 13 17 73 27 35 59 3 27 63 0 34 2 32 60 30 62 28
22* 12 46 14 44 16 42 18 40 47 11 15 43 19 39 57 1 25 33 29 29 58 0 60 26 26 32 28 30
23* 44 14 46 12 12 74 14 72 15 43 47 11 15 71 25 33 57 1 25 61 26 32 0 30 58 28 60 26
24* 14 48 16 46 18 44 20 42 49 13 17 45 21 41 59 3 27 35 31 31 60 2 62 0 28 34 30 32
25* 42 20 44 18 46 16 48 14 21 41 17 45 49 13 31 31 27 35 59 3 32 30 34 28 0 62 2 60
26* 16 14 18 12 12 46 14 44 15 15 19 11 15 43 25 5 29 1 25 33 26 4 28 2 30 0 32 26
27* 40 18 42 16 44 14 46 12 19 39 15 43 47 11 29 29 25 33 57 1 30 28 32 26 26 60 0 58
28* 18 16 20 14 14 48 16 46 17 17 21 13 17 45 27 7 31 3 27 35 28 6 30 4 32 2 34 0

```

```

*** 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
*****
1 * 0 4 1 6 3 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 17 17 28 25 19
2 * 10 0 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 23 13 27 15 24 21 26 17 28 25 19
3 * 10 4 0 6 3 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 17 17 28 25 19
4 * 10 4 1 0 12 8 5 14 2 16 3 18 7 20 9 22 11 23 13 27 15 24 21 26 17 28 25 19
5 * 10 4 1 12 0 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 26 17 28 25 19
6 * 10 4 1 6 12 0 5 14 2 16 6 18 7 20 9 22 11 23 13 27 15 24 21 17 17 28 25 19
7 * 10 4 1 12 12 8 0 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 26 17 28 25 19
8 * 10 4 1 6 12 8 5 0 2 16 6 18 7 20 9 22 11 23 13 27 15 24 21 17 17 28 25 19
9 * 10 4 1 12 12 8 5 14 0 16 3 18 7 20 9 22 11 23 13 27 15 24 21 26 17 28 25 19
10* 10 4 1 6 3 8 5 14 2 0 3 18 7 20 9 22 11 26 13 27 15 24 21 17 17 28 25 19
11* 10 4 1 6 3 8 5 14 2 16 0 18 7 20 9 22 11 23 13 27 15 24 21 17 17 28 25 19
12* 10 4 1 12 12 8 5 14 2 16 3 0 7 20 9 22 11 23 13 27 15 24 21 26 17 28 25 19
13* 10 4 1 12 12 8 5 14 2 16 3 18 0 20 9 22 11 26 13 27 15 24 21 26 17 28 25 19
14* 10 4 1 6 12 8 5 14 2 16 6 18 7 0 9 22 11 23 13 27 15 24 21 17 17 28 25 19
15* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 0 22 11 23 13 27 15 24 21 26 17 28 25 19
16* 10 4 1 6 3 8 5 14 2 16 3 18 7 20 9 0 11 26 13 27 15 24 21 17 17 28 25 19
17* 10 4 1 6 3 8 5 14 2 16 3 18 7 20 9 22 0 23 13 27 15 24 21 17 17 28 25 19
18* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 0 13 27 15 24 21 26 17 28 25 19
19* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 26 0 27 15 24 21 26 17 28 25 19
20* 10 4 1 6 12 8 5 14 2 16 6 18 7 20 9 22 11 23 13 0 15 24 21 17 17 28 25 19
21* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 23 13 27 0 24 21 26 17 28 25 19
22* 10 4 1 6 3 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 0 21 17 17 28 25 19
23* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 23 13 27 15 24 0 26 17 28 25 19
24* 10 4 1 6 3 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 0 17 28 25 19
25* 10 4 1 6 12 8 5 14 2 16 6 18 7 20 9 22 11 23 13 27 15 24 21 17 0 28 25 19
26* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 26 17 0 25 19
27* 10 4 1 6 12 8 5 14 2 16 6 18 7 20 9 22 11 23 13 27 15 24 21 17 17 28 0 19
28* 10 4 1 12 12 8 5 14 2 16 3 18 7 20 9 22 11 26 13 27 15 24 21 26 17 28 25 0

```

Shortest Path 7<--1:

7<--5<--3<--1

Shortest Path 11<--1:

11<--3<--1

Shortest Path 10<--1:

10<--16<--22<--24<--17<--11<--3<--1

Shortest Path 27<--1:

27<--25<--17<--11<--3<--1

LIST OF REFERENCES

- [BYR94]Byrne, P.G., "A Mobile Robot Sonar System with Obstacle Avoidance", Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1994.
- [KAN89]Kanayama, Y., Noguchi, T., "Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors", University of California Department of Computer Science *Technical Report* TRCS89-06, February, 1989.
- [KAN95a]Kanayama, Y., "Introduction to Motion Planning", *Lecture Notes of the Advanced Robotics Course*, Department of Computer Science, Naval Postgraduate School, March 25, 1995.
- [KAN95b]Kanayama, Y., "Theory of Path Classes for Robot Motion Planning", *Extended Abstract*, Department of Computer Science, Naval Postgraduate School, Monterey, California, November 7, 1995.
- [KAN95c]Kanayama, Y., Kovalchic, J.G., Chuang, C.-L., Kelbe, F.E., "Motion Planning for Autonomous Mobile Robots", Department of Computer Science, Naval Postgraduate School, Monterey, California.
- [KAN95d]Kanayama, Y., Kovalchic, J.G., Kelbe, F.E., "Motion Planning for Autonomous Mobile Robots" *Proceedings. Autonomous Vehicle in Mine Countermeasures Symposium pp. 8-74 to 8-80*, Monterey, California, April 4-7, 1995.
- [LOC94]Lochner J.T., "Analysis and Improvement of an Ultrasonic Sonar System on an Autonomous Mobile Robot", Master's Thesis, Naval Postgraduate School, Monterey, California, December, 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218
2. Dudley Knox Library.....2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101
3. Chairman, Code CS.....2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
4. Dr. Yutaka Kanayama, Code CS/KA.....2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
5. Dr. Xiaoping Yun, Code EC/YX.....1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943
6. Lt. Athanassios D. Papadatos.....2
107 Alexandras Ave
Athens, 11475 GREECE